



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

1987-12

A practical application of Petri nets in the software safety analysis of a real-time military system.

Hayward, Duston L.

<http://hdl.handle.net/10945/22423>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

MONTEREY, CALIF. CIVIL 95948 5003

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

H4C75

A PRACTICAL APPLICATION OF PETRI NETS
IN THE SOFTWARE SAFETY ANALYSIS
OF A REAL-TIME MILITARY SYSTEM

by

Duston L. Hayward

December 1987

Thesis Advisor:

Daniel Davis

Approved for public release; distribution is unlimited.

T238969

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited.	
DECLASSIFICATION / DOWNGRADING SCHEDULE		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
PERFORMING ORGANIZATION REPORT NUMBER(S)		7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b. OFFICE SYMBOL (If applicable) Code 52	
ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	
NAME OF FUNDING / SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	
ADDRESS (City, State, and ZIP Code)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
TITLE (Include Security Classification) PRACTICAL APPLICATION OF PETRI NETS IN THE SOFTWARE SAFETY ANALYSIS OF A REAL-TIME MILITARY SYSTEM (u)		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
PERSONAL AUTHOR(S) Howard, Duston L.			
TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 December	15. PAGE COUNT 117
SUPPLEMENTARY NOTATION			
COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
ABSTRACT (Continue on reverse if necessary and identify by block number) <p>This thesis evaluates the usefulness of Petri net modeling for software safety analysis of a real-time system. The system is a safety arming device for a guided missile. The features of basic Petri net modeling are discussed in relation to the kinds of components that are found in real-time systems. This thesis proposes a methodology for systematically constructing a Petri net model from system and software design information. Several techniques for analyzing the resulting Petri net model are illustrated and evaluated for appropriateness.</p>			
DISTRIBUTION / AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
NAME OF RESPONSIBLE INDIVIDUAL Prof. Daniel Davis		22b. TELEPHONE (Include Area Code) (408) 646-3091	22c. OFFICE SYMBOL Code 52Dv

Approved for public release; distribution is unlimited

A Practical Application of Petri Nets
in the Software Safety Analysis of a
Real-Time Military System

by

Duston L. Hayward
B.S., The Pennsylvania State University, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the .

NAVAL POSTGRADUATE SCHOOL
December 1987

ABSTRACT

This thesis evaluates the usefulness of Petri net modeling for software safety analysis of a real-time system. The system is a safety arming device for a guided missile. The features of basic Petri net modeling are discussed in relation to the kinds of components that are found in real-time systems. This thesis proposes a methodology for systematically constructing a Petri net model from system and software design information. Several techniques for analyzing the resulting Petri net model are illustrated and evaluated for appropriateness.

TABLE OF CONTENTS

Thesis
44695
c.1

I.	INTRODUCTION	6
II.	INTRODUCTION TO SOFTWARE SAFETY	9
III.	REAL-TIME SYSTEM DEFINED	23
IV.	PETRI NET THEORY AND MODELING METHODOLOGY	28
V.	SOFTWARE HAZARD ANALYSIS, RESULTS OF ANALYSIS, AND METHODOLOGY	65
VI.	CONCLUSIONS AND RECOMMENDATIONS	91
APPENDIX A	SOFTWARE CODE LISTING	94
APPENDIX B	SOFTWARE FLOWCHART	98
APPENDIX C	SYSTEM FLOWCHART	100
APPENDIX D	SYSTEM ELEMENTS	101
APPENDIX E	PETRI NET OF SYSTEM ELEMENTS	102
APPENDIX F	SOFTWARE ELEMENTS	104
APPENDIX G	PETRI NET OF SOFTWARE ELEMENTS	105
APPENDIX H	PETRI NET MODEL	106
APPENDIX I	REACHABILITY GRAPH THEORY	109
APPENDIX J	CRITICAL STATE DETERMINATION ALGORITHM	111
	LIST OF REFERENCES	113
	INITIAL DISTRIBUTION LIST	115

ACKNOWLEDGMENT

I would especially like to thank my thesis advisor, Professor Daniel Davis for his hours of help and understanding. I would also like to thank Steve Rohde for helping me to understand a guided missile system and to complete my model of the system. I would like to thank Nancy Leveson, University of California-Irvine, for laying the foundation for my research in the software safety area. I would also like to thank Robert Westbrook, China Lake Naval Weapons Center, who suggested that I investigate this field and my second reader Professor Gordon Bradley.

I. INTRODUCTION

"Computers are increasingly being used to monitor and/or control complex, time-critical physical processes or mechanical devices, where a run-time error or failure could result in death, injury, loss of property, or environmental harm." [Leveson 1986] As the use of computer software in weapons systems grows, computer scientists and system engineers are now faced with the difficult and unsolved problems concerning the safety of software used in these systems.

There are increasing requirements for building safety-critical systems and the military is establishing standards that must be met for safety in these systems. Such standards as MIL-STD-882B, MIL-STD, 1574A, MIL-STD-SNS already include software related requirements, such as software-hazard analysis, and verification of software safety. Such techniques as software fault tree analysis, software sneak analysis, and Petri net analysis are listed in MIL-STD-882B Notice 1 of 1 July 1987 as techniques available to perform software safety analysis.

The question is whether the software engineering community is ready to respond to these requirements for software safety. Current software reliability enhancing techniques and software

reliability models do not satisfy these requirements. New techniques and approaches are needed, along with new perspectives and emphasis.

This thesis will briefly investigate alternative techniques to perform this type of analysis, but then focus on if the requirements of these standards can be met by using a Petri net methodology. The objective of this thesis is to investigate the feasibility of using Petri nets to model and analyze real time systems and if it is found feasible to propose a usable methodology for performing this work. Nancy Leveson of the University of California - Irvine has proposed the use of Petri nets as a modeling tool and has provided a set of techniques to perform software safety analysis on the model. This thesis work is the first known application of the technique proposed by Leveson. For more information on the work done by Leveson the reader should see [Leveson 1986; Leveson and Stolzy 1987].

The system under evaluation is a mechanical safety arming device for a guided missile mechanical fuze. Microprocessors now control arming and firing of fuzes, weapon release, navigation and control of missiles. In the system analyzed, in an attempt to save cost in manufacturing and improve reliability, a microprocessor and software are replacing part of a safety arming device that was previously mechanical.

Microprocessors allow for expanded control in addition to added capability with less space and weight [McIntee 1983].

This thesis begins with a discussion of what software safety is and what it isn't. Then the motivation for using Petri nets is discussed and alternative techniques are reviewed. This discussion includes various techniques that have also been considered to perform this analysis and ones that have been tried in the past. That is followed by the theory and modeling technique of Petri nets to model a real-time system. Then the actual real time system is defined. Then the methodology used to perform the analysis is discussed, followed by the results of the analysis and recommendations for further research.

II. INTRODUCTION TO SOFTWARE SAFETY

A. SOFTWARE SAFETY DEFINED

Safety is usually considered to be a part of either reliability or security. In order to adequately address the specific requirements of safety, safety must be considered a separate area. There are some aspects of software safety that are unique with respect to current software engineering concepts. Leveson [1986] argues separate consideration of safety allows special emphasis and separation of concerns on how decisions are being made. To ensure that the final system is safe, it is necessary to make explicit any trade-offs that involve safety. Safety requirements should be separated and identified and responsibilities should be assigned. [Leveson 1986]

1. Reliability versus Safety

Safety and reliability are often equated, especially with respect to software, but the concepts, though related, are not the same. Reliability is usually defined as the probability that a system will perform its intended function for a specified period of time under a set of specified environmental conditions. Safety is the probability that conditions that can lead to a mishap (hazards) do not occur, whether or not the intended function is performed [Ericson 1981; Konakovsky 1978; Leveson 1986]. In general, reliability

requirements are concerned with making a system failure free, whereas safety requirements are concerned with making it mishap free. Reliability is concerned with every possible software error, whereas safety is only concerned with those that result in actual system hazards. Not all software errors cause safety problems, and not all software that functions according to specifications is safe [Ericson 1981]. Severe mishaps have occurred while something was operating exactly as intended, that is, without failure [Roland and Moriarity 1983]. [Leveson 1986]

System requirements can be separated into those related to the mission and those related to safety while the mission is being accomplished. For munitions, reliability is the probability of detonation or functioning of the munition at the desired time and place, while safety is related to inadvertent functioning, so there is no direct relationship [Leveson 1986]. Procedures to increase the ability of the weapon to fire when desired may increase the likelihood of accidental detonation, unless the design of the munition is modified to improve the safety as the reliability increases [Roland and Moriarity 1983].

Leveson and Stolzy [1987] have argued that there is a need for a completely different approach to safety problems, that is, an approach that is complementary to standard reliability techniques and focuses on the failures that have the most drastic consequences. "Even if all failures cannot

be prevented, it may be possible to ensure that the failures that do occur are of minor consequence or that, even if a potentially serious failure does occur, the system will fail-safe." [Leveson and Stolzy 1987] Fail-safe procedures attempt to limit the amount of damage caused by a failure and there is no attempt to satisfy the functional specifications except where necessary to ensure safety.

This approach is useful when not all failures are of equal consequence and there is a relatively small number of failures that can lead to catastrophic results. Under these circumstances, it is possible to augment traditional reliability techniques that attempt to eliminate all failures with techniques that concentrate on the high-cost failures. These new techniques often involve a "backward" approach that starts with determining what are unacceptable or high-cost failures and then ensures that these particular failures do not occur or at least minimizes the probability of their occurrences. [Leveson 1986]

In system engineering, reliability and safety are usually distinguished. After some significant mishaps, system safety has begun to receive more attention with stricter standards being issued and enforced. When software constitutes an important part of a safety-critical system software safety needs to be given the same attention.

2. Safety versus Security

Safety and security are closely related. Both deal with threats or risks, one with threats to life or property and the other with threats to privacy or national security. Both often involve negative requirements that may conflict with some important functional or mission requirements. Both involve requirements that are considered of supreme importance in deciding whether the system can and should be used.

[Leveson 1986]

Security focuses on unauthorized actions, whereas safety is more concerned with inadvertent actions. The primary emphasis in security research has been on preventing unauthorized access to classified information.

3. A Systems Approach

It has been argued that there is no such thing as software safety since software cannot, by itself, be unsafe. A broader system view is that software can have various unexpected and undesired effects when used in a complex system [Dean 1981]. Software is correct or incorrect only with respect to some larger system in which it is functioning.

Safety must be defined in terms of hazards or states of the system that when combined with certain environmental conditions could lead to a mishap. Risk is a function of the probability of the hazardous state occurring, the probability of the hazard leading to a mishap, and the perceived severity of the worst potential mishap that could result from the

hazard. Thus there are two aspects of risk: 1) the probability of the system getting into a hazardous state and 2) the probability of the hazard leading to a mishap combined with the severity of the resulting mishap. System hazards may be caused by hardware component failure, design faults in the hardware or software, interfacing problems between components of the system, human error in operation or maintenance, or environmental problems. [Leveson 1986]

An accident is traditionally defined by safety engineers as an unwanted and unexpected release of energy [Johnson 1973]. Mishap is an unplanned event or series of events that result in death, injury, occupational illness, damage to or loss of equipment or property. [Leveson 1986]

Mishaps are almost always caused by multiple factors, and the relative contribution of each factor is usually not clear. A mishap can be thought of as a set of events combining in a random fashion [Petersen 1971], or alternatively, as a dynamic mechanism that begins with the activation of a hazard and flows through the system as a series of events in a logical sequence until the system is out of control and a loss is produced [Malasky 1982].

The state of the system is made up of the states of the components of the system, one of which is the computer. Often the computer acts as the controller of the system and thus has a direct effect on the current state. Therefore it makes sense to talk about "software safety" since the software

usually has at least partial control over whether the system is in a hazardous state or not. [Leveson 1986]

Software safety then involves ensuring the software will execute within a system context without resulting in unacceptable risk. What risk is acceptable or unacceptable must be defined for each system. As with "hardware safety", software safety is achieved by identifying potential hazards early in the development process and then establishing requirements and design features to eliminate or control these hazards [Ericson 1981]. Safety-critical software functions are those that can directly or indirectly cause or allow a hazardous system state to exist. [Leveson 1986]

4. Why Does a Problem Exist ?

"Many of the system safety techniques that have been developed to aid in building electromechanical systems with minimal risk do not seem to apply when computers are introduced. The major differences appear to stem from the differences between hardware and software and from the lack of system-level approaches to building software controlled systems." [Leveson 1986]

System safety techniques are designed to cope primarily with random failures in these systems. Human design errors are not considered since it is assumed that all faults caused by human errors can be avoided completely or located and removed prior to delivery and operation [Lauber 1980].

The advent of microprocessors and powerful automation procedures have dramatically increased the complexity of software and hardware, causing a nonlinear increase in human-error-induced design faults. Because of this complexity, it appears to be impossible to demonstrate that the design of the computer hardware or software of a realistic control system is correct and that failure mechanisms are completely eliminated [Lauber 1980]. By using computers to control processes, we are increasing both of these factors and, therefore increasing the potential for problems.

An important difference between conventional hardware control systems and computer-based control systems is that hardware has historical usage information, whereas control software usually does not [Gloe 1979]. Hardware is usually produced in greater quantities than software, and standard components are reused frequently. Therefore reliability can be measured and improved through experience in other applications. Software, on the other hand, is almost always specifically constructed for each application.

Not only are exhaustive testing and analysis impossible for most nontrivial software, but it is difficult to provide realistic test conditions. Most testing must be done in a simulation mode, and there is no way to guarantee that the simulation is accurate. Assumptions must always be made about the controlled process and its environment [Leveson 1986]. For example, the limits on the range of control

imposed by the software for the F/A-18 aircraft are based on assumptions about the ability of the aircraft to achieve certain attitudes, but unfortunately, a mishap occurred because an intentionally excluded attitude was actually attainable [Neumann 1981]. A wing-mounted missile on the F/A-18 failed to separate from the launcher after ignition because a computer program signaled the missile-retaining mechanism to close before the rocket had built up sufficient thrust to clear the missile from the wing [Frola and Miller 1984]. An incorrect assumption had been made about the amount of time that this would take. The aircraft went out of control. [Leveson 1986]

"These types of problems are not caught by the usual simulation process since they either have been considered and discarded as unreasonable or involve a misunderstanding about the actual operation of the process being controlled by the computer." [Leveson 1986] After studying serious mishaps related to computers, system safety engineers have concluded that inadequate design foresight and specification errors are the greatest cause of software safety problems [Ericson 1981; Griggs 1981]. Testing can show the consistency only with the requirements as specified; it cannot identify misunderstandings about the requirements. These can be identified only by use of the software in the actual system, which can, of course, lead to mishaps. Also, accurate live

testing of computer responses to catastrophic situations is of course, difficult in the absence of accidents [Leveson 1986].

"The point in time or environmental conditions under which the computer fault occurs may determine the seriousness of the result. Software faults may not be detectable except under just the right combination of circumstances, and it is difficult, if not impossible, to consider and account for all environmental factors and all conditions under which the software may be operating." [Leveson 1986]

To complicate things even further, most verification and validation techniques for software assume "perfect" execution environments. "However software failures may be caused by such undetected hardware errors as transient faults causing mutilation of data, security violations, human mistakes during operation and maintenance, errors in underlying or supporting software, or interface problems with other parts of the system such as timing errors. It is difficult, if not impossible, to test the software under all failure modes of the system. Trying to include all of these factors in the analysis or testing procedures makes the problem truly impossible to solve, given today's technology." [Leveson 1986]

For hardware, redundancy can be used to provide fault tolerance, since either the individual components can be shown to fail independently, or common-mode analysis techniques can detail dependent failure modes and minimize them. A similar

application of redundancy has been proposed for software [Anderson and Lee 1981; Avizienis 1985]. There is no evidence that the high reliability required in safety-critical software can be achieved using these techniques. Perhaps the most important consideration is that most fault-tolerance methods do not solve the problem of erroneous requirements [Leveson 1986].

The greatest cause of the problems experienced when computers are used to control complex processes may be a lack of system-level methods and viewpoints. Many hardware-oriented system engineers do not understand software because of the youth of software engineering and the significant differences between software and hardware [Ericson 1981]. The same is true, in reverse, for software engineers. This has led system engineers to consider the computer as a black box [Griggs 1981; Kletz 1983; Software Safety Handbook], whereas the software engineer has treated the computer as merely a stimulus-response system [Alford 1985 and Davis 1982]. This lack of communication has been blamed for several mishaps. [Leveson 1986]

An obvious conclusion is that system-level approaches are necessary [Boebert 1980; Lauber 1980; Leveson and Stolzy 1987]. In fact, it is difficult to define a software "fault" without considering the system. If the problem stems from an error in the requirements, then the software may be "correct" with respect to the stated software requirements, but wrong

from a system standpoint. A particular software fault may cause a mishap only if there is a simultaneous human and/or hardware failure. "Software engineering techniques that do not consider the system as a whole, including the interactions between the hardware, software, and human operators, will have limited usefulness for real-time control software." [Leveson 1986]

Traditional software engineering techniques tend to consider the software in a vacuum. But examining the software separately does not provide information about interfacing problems between the software and other components of the system. By looking at the software alone, it is impossible to examine problems which occur only as a result of an interaction between software and a failure of another system component. [Leveson and Stolzy 1983]

The software must be analyzed within the context of the entire system, including the computer hardware, the other components of the system, and the environment. Software as a black box in a system is no longer a valid approach. A total system concept must be considered within the analysis so the effects of the software on the whole system can be taken into account. [Leveson 1986]

B. TECHNIQUES FOR SOFTWARE SAFETY REQUIREMENTS ANALYSIS

Determining the requirements for software has proven to be very difficult. Besides timed Petri nets [Leveson and Stolzy

1987]], several techniques have been proposed and used in limited contexts, including fault tree analysis and real-time logic. This is a major source of software problems and may be the most important with respect to safety.

While functional requirements often focus on what the system shall do, safety requirements must also include what the system shall not do - including means of eliminating and controlling system hazards and of limiting damage in case of a mishap. An important part of the safety requirements is the specification of the ways in which the software and the system can fail safely and to what extent failure is tolerable.

[Leveson 1986]

1. Fault Tree Analysis

Fault Tree Analysis [Vesely et al. 1981] is an analytical technique used in the safety analysis of electromechanical systems. An undesired system state is specified and the system is then analyzed in the context of its environment and operation to find believable sequences of events that can lead to the undesired state. The fault tree is a graphic model of various parallel and sequential combinations of faults that will result in the occurrence of the unwanted event. A fault tree depicts the logical interrelationships of basic events that can lead to the hazardous event. [Leveson 1986]

"The success of the technique is highly dependent on the ability of the analyst, who must thoroughly understand the

system being analyzed and its underlying scientific principles. However, it has the advantage that all of the system components can be considered. This is extremely important because a particular software fault may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, the environmental failure may cause the software fault to manifest itself." [Leveson 1986]

The analysis starts with a list of system hazards that have been identified by a preliminary hazard analysis (PHA). A separate fault tree must be constructed for each hazardous event. The basic procedure is to assume that the hazard has occurred and then to work backward to determine the set of possible causes.

Software control faults may include: 1) failure to perform a required function, 2) performing a function not required, 3) timing or sequencing problems, 4) failure to recognize a hazardous condition requiring corrective action, and 5) producing the wrong response to a hazardous condition.

As the development of the software proceeds, fault tree analysis can be performed on the design and finally the actual code. Software fault tree analysis can be used to determine software safety requirements, detect software logic errors, identify multiple failure sequences involving different parts of the system (hardware, human, and software) that can lead to hazards, and guide in the selection of critical run-time checks. It can also be used as a guide for

testing. The interfaces of the software parts of the fault tree can be examined to determine appropriate test input data and appropriate simulation states and events. [Leveson 1986] For further information on this technique the reader is referred to [Leveson and Stolzy 1983] and [McIntee 1983].

2. Real-Time Logic

Jahanian and Mok [1986] have shown how to formalize the safety analysis of timing properties in real-time systems using a formal real-time logic (RTL). The system designer first specifies a model of the system in terms of events and actions.

To analyze the system design, the RTL formulas are translated into predicates of Presburger arithmetic with uninterpreted integer functions. Decision procedures are then used to determine whether a given safety assertion is a theorem derivable from the system specification. If so, the system is safe with respect to the timing behavior denoted by that assertion as long as the implementation satisfies the requirements specification. If the safety assertion is unsatisfiable with respect to the specification, then the system is inherently unsafe because successful implementation of the requirements will cause the safety assertion to be violated. A restricted set of Presburger formulas are used that allow for a more efficient decision procedure. [Leveson 1986]

III. A SAMPLE REAL-TIME SOFTWARE SYSTEM

A. CHOICE OF A SYSTEM

In choosing a system to evaluate Petri net modeling for software safety analysis it was necessary to find a system that would be small and at the same time representative of a number of systems. The system needed to be small enough to build the model, perform the analysis and write about it in a limited period of time. For a system to be representative it needed to be a real-time system with interacting components and be a subsystem of a total system with software and hardware concerns as well as environmental concerns. If a real life system could not be found, for research purposes a sample system would have to be created.

Fortunately, there was a need for a software safety analysis on a real life problem at The Naval Weapons Center in China Lake, California. After a meeting to discuss the system and a preliminary analysis, the system was determined to be small enough in size to be tackled in a short period of time and to be able to make an accurate assessment of the technique.

The system that was chosen was an interrupted-explosive train safety-arming (SA) device for a guided-missile. This project was a new concept being investigated by the Navy to

use software and a microcomputer to replace a mechanical device to perform the safe separation distance calculation for a safety arming device. This project could potentially have a tremendous cost saving to the Navy and would provide a state-of-the-art way of doing business.

A software prototype of the system was written and tested, but without safety considerations. For a complete listing of the software see Appendix A. Since the safety considerations had not been fully considered this system should be ideal for evaluating the effectiveness of the Petri net technique.

B. SYSTEM BACKGROUND

A safety arming device is a precision safety item incorporating mechanical, electronic, and explosive components. The function of the safety arming device is to prevent inadvertent high-explosive warhead initiations throughout the logistic cycle of a weapon, with a high degree of confidence, and to arm at the correct point in tactical use for desired warhead initiation. [McVay 1987]

The safety features for the design of a safety arming device are contained in [MIL-STD-1316C]. There must be two independent safety features: 1) to prevent unintentional arming and 2) to provide forces to remove safety features derived from different environments. Operation of at least

one of these safety features shall depend on sensing a post-launch environment.

Safety features shall also provide an arming delay. The arming delay shall provide safe separation distance for all defined operational conditions. Also, an assembled fuze shall not be capable of being armed manually.

In the strictly mechanical systems, the SA device contains a mechanism that is mechanically locked in the "safe" position until it is unlocked by application of an electrical current to a solenoid. When the rocket motor in the guided missile is fired, missile acceleration drives a setback weight that is connected to a rotor containing the interrupted element through the gears of an escapement mechanism. The escapement acts as a pseudo-integrator which facilitates moving the rotor from the "interrupted" or "safe" position to the "armed" position when the missile has traveled a preset safe distance from the launcher. [McVay 1987]

A block diagram of a generic non-interrupted explosive train SA device for a boosted guided missile is shown in Figure 2-1. The block diagram was specifically formulated for a guided missile SA device and therefore contains two interruptors. One interruptor is in the low voltage section and one is in the high voltage section. The interruptors are directly and mechanically locked as required by MIL-STD-1316C

and the interruptors are removed by energy from a unique post-launch environment. [McVay 1987]

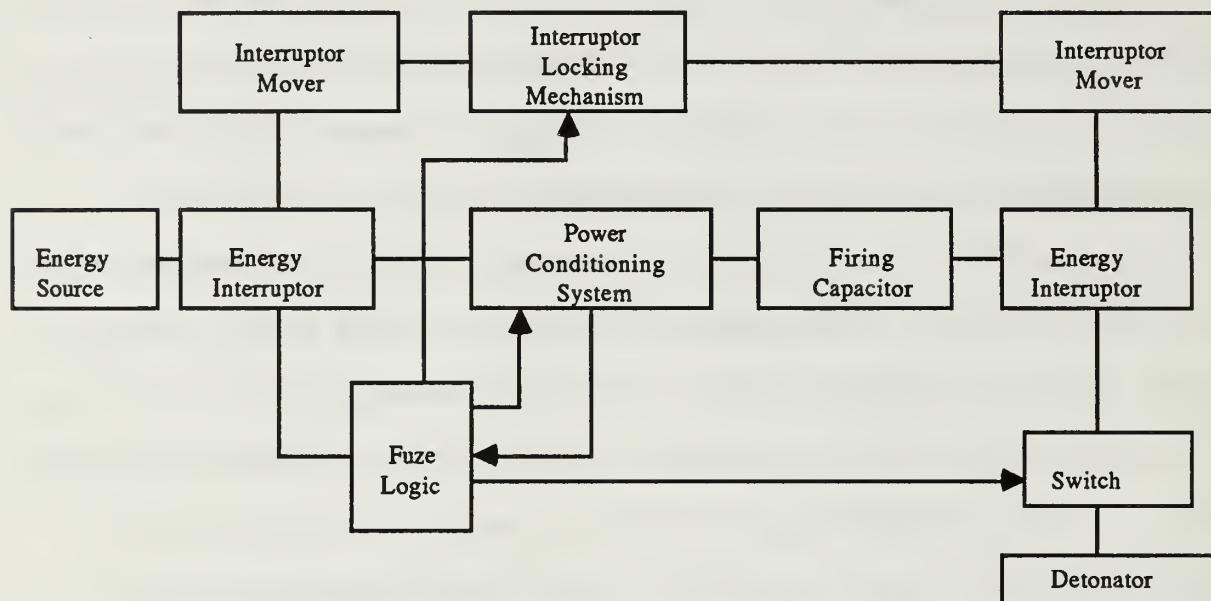


Figure 2-1. Boosted Guided Missile Noninterrupted Explosive Train SA Device

In the prototype system with a microcomputer and software, the scenario starts with the missile on the aircraft rack. The aircraft sends an intent to launch, which sends power to the computer and software. This also begins to charge the firing capacitor. The intent to launch unlocks the safety arming device and fires the thermal battery. The rocket motor fires and the missile launches. A 4+ G boost occurs and the safety arming device begins to compute safe separation distance. The software goes through three iterations where it reaches a specific distance and toggles a solenoid at each preset

distance. Each solenoid toggle activates a ball locking mechanism and once three toggles have occurred the mechanical detonator interruptor is removed. Now the safety arming device arms and when target detection occurs a signal is sent to detonate the warhead.

The software for this system resides on a Intel 8048 silicon chip microcomputer and is written in 8048 assembly language. The 8048 is a 40 pin package containing:

1) an 8-bit CPU, 2) 1K x 8 ROM program memory, 3) 64 x 8 RAM data memory, 4) 27 I/O lines, and 5) 8-bit timer/event counter. It also has a 2.5 or 5.0 microsecond cycle time and 90 instruction instruction-set [Intel 1978].

IV. PETRI NET THEORY AND MODELING METHODOLOGY

A. WHY PETRI NETS?

Petri nets have been developed from the early work of Carl Adam Petri, "Kommunikation mit Automaten" [Petri 1962]. Petri formulated the basis for a theory of communication between asynchronous components of a computer system. A.W. Holt and others studied the work of Petri and much of the early theory, notation, and the representation of Petri nets developed from their work. This work showed how Petri nets could be applied to the modeling and analysis of systems of concurrent components.

Petri nets were designed for and are used mainly for modeling. Many systems, especially those with independent components can be modeled by a Petri net.

Petri nets have been used to model and analyze systems for such properties as deadlock and reachability. In this thesis the goal is to show whether or not Petri nets can be used to adequately model a system for software safety analysis. A systems approach is valid since hardware, software, and human behavior can be modeled all in one net.

Before providing a formal definition for using Petri nets to model real time systems, the motivation for using this

method is discussed. A detailed description of the elements in a Petri net model will be discussed in an upcoming section.

1. Parallelism

Petri nets are able to model parallelism and concurrency. Consider the case of two processes. Each process can be represented by a Petri net. Thus the composite Petri net, which is the union of the Petri nets for each of the two processes can represent the concurrent execution of the two processes.

An example, shown in Figure 4-1, is the FORK and JOIN operations originally proposed by Dennis and Van Horn [1966]. A FORK j operation executed at location i results in the current process continuing at location $i + 1$ and a new process being created with execution started at location j . A JOIN operation will recombine two processes into one (or equivalently will destroy one of the two and let the other proceed). [Peterson 1981]

2. Synchronization

The sharing of resources and information between processes must be controlled to ensure correct operation of the overall system. Petri nets can also model synchronization. A synchronization problem which illustrates the types of problems which can arise between cooperating processes is the mutual exclusion problem [Dijkstra 1965], presented in Figure 4-2.

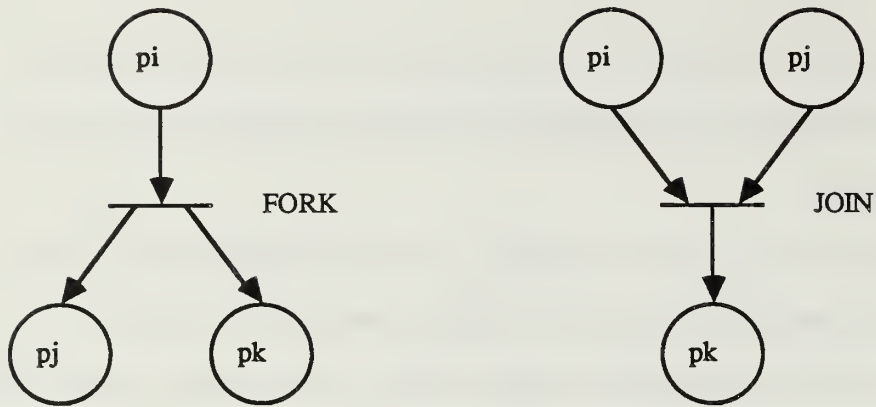


Figure 4-1. Modeling FORK and JOIN with Petri nets

In mutual exclusion, access to the critical sections of the two processes is controlled so that both processes cannot simultaneously execute their critical sections.

In this Petri net, place *m* represents the permission to enter the critical section. For a process to enter the critical section, it must have a token in *p1* or *p2*, as appropriate, signalling that it wishes to enter the critical section, and there must be a token in place *m* signalling permission to enter. If both processes wish to enter simultaneously, then transitions *t1* and *t2* are in conflict, and only one of them can fire. Firing *t1* will disable transition *t2*, requiring process 2 to wait until the first process exits its critical section and puts a token back in place *m*. [Peterson 1981]

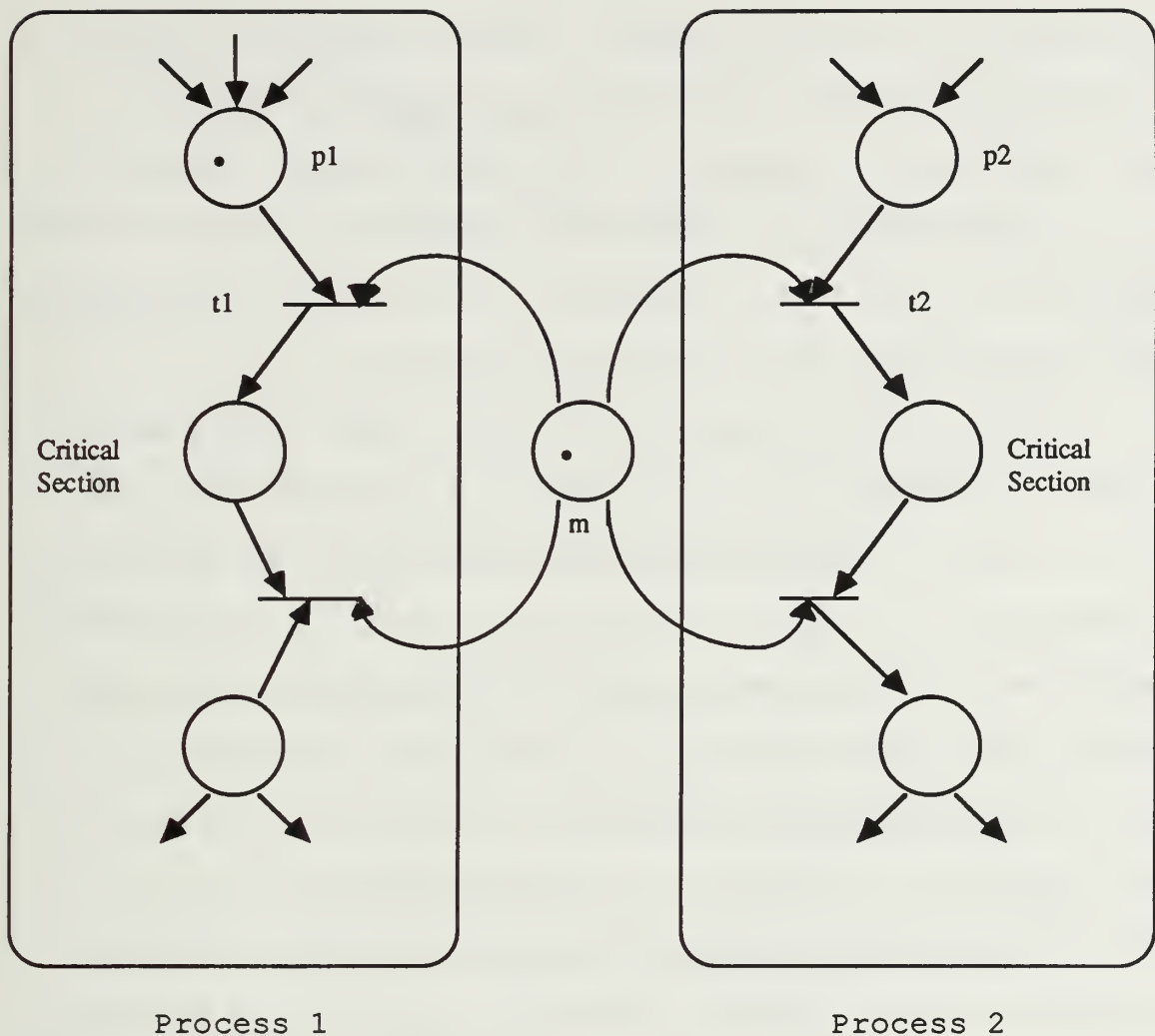


Figure 4-2. Mutual exclusion modeled with Petri nets

3. Software Hazard Analysis

Petri net models have been proposed for software hazard analysis. Petri nets allow mathematical modeling of discrete-event systems in terms of conditions and events and the relationship between them. Analysis and simulation procedures have been developed to determine desirable and undesirable properties of the design, especially with respect

to concurrent or parallel events. Leveson and Stolzy [1987] have developed analysis procedures to determine software safety requirements (including timing requirements) directly from the system design, to analyze a design for safety, recoverability, and fault tolerance, and to guide in the use of failure detection and recovery procedures.

Faults and failures can be incorporated into the Petri net model to determine their effects on the system [Leveson and Stolzy 1987]. Backward analysis procedures can be used to determine which failures and faults are potentially the most hazardous and therefore which parts of the system need to be augmented with fault-tolerance and fail-safe mechanisms. Early in the design of the system it is possible to treat the software parts of the design at a very high level of abstraction and consider only failures at the interfaces of the software and nonsoftware components. By working backward to the software interfaces, it is possible to determine the software safety requirements and identify the most critical functions. [Leveson 1986]

Now that part of the motivation for use of Petri nets has been provided, a formal definition of Petri nets is provided using the notation of Peterson [Peterson 1981].

B. PETRI NET THEORY

1. Petri Net Structure

definition: Petri net structure, Φ , is a 5-tuple,

$$C = (P, T, I, O, \mu).$$

$P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places, $n \geq 0$.

$T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions, $m \geq 0$.

The set of places and the set of transitions are disjoint, $P \cap T = \emptyset$.

$I: T \rightarrow P^\infty$ is the input function, a mapping from transitions to bags of places.

$O: T \rightarrow P^\infty$ is the output function, a mapping from transitions to bags of places.

$\mu_0: P \rightarrow N$ is the initial marking for the net where N is the set of nonnegative integers.

A bag is a generalization of a set that allows multiple occurrences of an element. The operations on bags are union, intersection, sum, and difference.

As an example, consider the followings bags over $\{a, b, c, d\}$:

$$B_1 = \{a, b, c\}$$

$$B_2 = \{a\}$$

$B_3 = \{a, b, c, c\}$

$B_4 = \{a, a, a\}$

The use of bags for the inputs and outputs of a transition allows a place to be a multiple input or a multiple output of a transition.

definition: The multiplicity of an input place p_i for a transition t_j is the number of occurrences of the place in the input bag of the transition, $\#(p_i, I(t_j))$.

I and O can be extended to map places into bags of transitions in addition to mapping transitions into bags of places.

2. Petri Net Graphs

A Petri net graph is a representation of a Petri net structure as a bipartite directed multigraph.

A Petri net has two types of nodes:

- a circle \bigcirc represents a place
- a bar $|$ represents a transition

Directed arcs connect the places and the transitions. Multiple inputs to a transition are indicated by multiple arcs from the input places to a transition.

definition: a transition t_j is an input of place p_i if p_i is an output of t_j .

$$\#(t_j, I(p_i)) = \#(p_i, O(t_j))$$

$$I: P \rightarrow T^\infty, O: P \rightarrow T^\infty$$

In Figure 4-3, t_1 represents a transition that is an input of place P_1 and place P_1 is an output of transition t_1 .

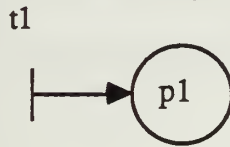


Figure 4-3. Transition t_1 is an input to place P_1

definition: a transition t_j is an output of place p_i if p_i is an input of t_j .

$$\#(t_j, O(p_i)) = \#(p_i, I(t_j))$$

In Figure 4-4, t_1 represents an output of place P_1 and place P_1 is an input of transition t_1 .

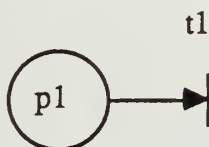


Figure 4-4. Transition t_1 is an output of place P_1

3. Execution Rules for Petri Nets

The execution of a Petri net is controlled by the number and distribution of tokens in the net. Tokens reside in the places and control the execution of the transitions of the net.

A Petri net executes by firing transitions. A transition fires by removing tokens from its input places and then depositing into each of its output places one token from each arc from the transition to the place. A transition may fire only if it is enabled.

definition: a transition is enabled, as in Figures 4-5, 4-6, and 4-7, if each of its input places has at least as many tokens in it as arcs from the place to the transition. Multiple tokens are needed for multiple input arcs. The tokens in the input places which enable a transition are its enabling tokens.

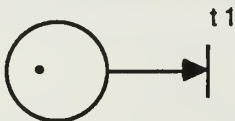


Figure 4-5. Transition t1 is enabled.

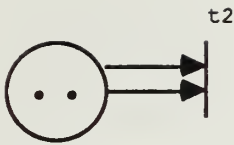


Figure 4-6. Transition t2 is enabled

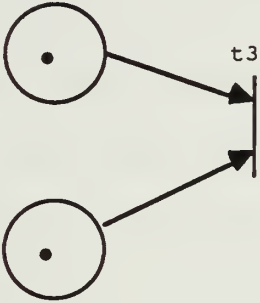


Figure 4-7. Transition t3 is enabled

In Figure 4-8, transition t4 is not enabled, since there are two inputs to enable t4 and there is only one token in the place.

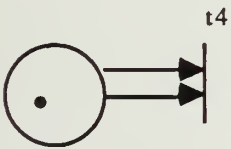


Figure 4-8. Transition t4 is not enabled

Figures 4-9 and 4-10 show how multiple tokens are produced from the firing of a transition with multiple output arcs.

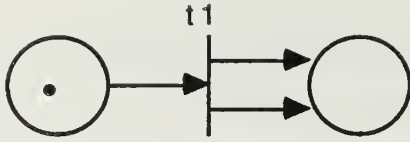


Figure 4-9. T1 is enabled

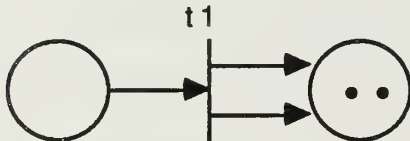


Figure 4-10. After t1 fires

In the following example, shown in Figures 4-11 and 4-12, the transition has multiple input arcs, with only a single output arc. This produces only one output token from the transition.

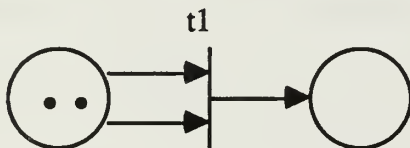


Figure 4-11. Transition t1 is enabled

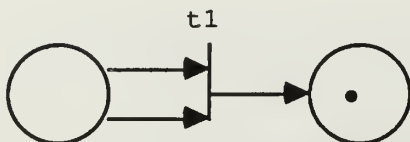


Figure 4-12. After t1 fires

Transition firings can continue as long as there exists at least one enabled transition. When there are no enabled transitions, the execution halts.

4. Petri Net State Spaces

The state of a Petri net is defined by its markings. For example, if a token is present in each one of these places: P_1, P_3, P_5 , then $(P_1 P_3 P_5)$ represents the state of the net. The change in state caused by firing a transition is defined by the next-state function δ .

definition: The next state function $\delta: N^n \times T \rightarrow N^n$ for a Petri net $\Phi = (P, T, I, O, \mu_0)$ with marking μ and transition $t_j \in T$ is defined if and only if t_j is enabled.

If $\delta(\mu, t_j)$ is defined, then $\delta(\mu, t_j) = \mu'$ where $\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$ for all $p_i \in P$.

5. Time Petri Nets

In this research, the Time Petri nets are not used, but the approach used is the traditional concept of Petri nets. There are some notations made concerning the timing constraints of the system and the timing constraints are considered. Safety concepts not concerning timing constraints can be considered with the use of traditional nets.

The information concerning Time Petri nets is included as an assistance to the reader, since an application might be

concerned with critical timing constraints. In either approach, it is important first to build the untimed Petri net model and then to add the timing constraints.

To model real time requires enhancements to the traditional Petri net model. There have been several proposals for extending standard Petri nets to include time. Leveson and Stolzy [1987] chose the Merlin and Farber [1976] approach where min and max times define a range of delays for each transition. Tokens are allowed to remain on the input places during the transition delay so the model retains the instantaneous firing features of untimed Petri nets while also providing a very flexible modeling tool.

A Time Petri net is a Petri net plus the added firing time functions min and max. The firing time functions specify the conditions under which a transition may fire.

A formal definition of a Time Petri Net structure is:

$$\Phi = (P, T, I, O, \text{Min}, \text{Max}, \mu_0).$$

$P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places, $n \geq 0$.

$T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions, $m \geq 0$.

The set of places and the set of transitions are disjoint,
 $P \cap T = \emptyset$.

$I: T \rightarrow P^\infty$ is the input function, a mapping from transitions to bags of places.

$O: T \rightarrow P^\infty$ is the output function, a mapping from transitions to bags of places.

Min and Max are the min time function and max time function, respectively, where:

$\text{Min}: T \rightarrow R$ and $\text{Max}: T \rightarrow R$, R is the set of nonnegative real numbers and $\text{Min}_i \leq \text{Max}_i$ for all i such that $t_i \in T$.

$\mu_0: P \rightarrow N$ is the initial marking for the net where N is the set of nonnegative integers.

definition: A transition t_j is firable at time τ if and only if it has been continuously enabled during the interval $\tau - \text{Min}(t_j)$ to τ . The firable transition may fire at any time τ for $\text{Min}(t_j) \leq \tau \leq \text{Max}(t_j)$. A transition must fire at time τ if it has been continuously enabled during the interval $\tau - \text{Max}(t_j)$ to τ .

definition: The state of the net σ consists of the tuple (μ, E) where μ is the marking and E is the remaining enabling time vector.

E is a function of a set of tuples of real numbers R , $E: (R \times R) \rightarrow (R \times R)$.

The timed Petri nets are more complex because of the continuous nature of time. Since transitions may fire at any time in their allowed interval, the states have in general an

unbounded number of successors [Leveson and Stolzy 1987]. Berthomieu and Menasche [1983] solve this problem by defining state classes that consider the set of all states reachable from the initial state by a given sequences of transitions.

The Time Petri net is equivalent to a standard Petri net if all Min times are 0 and all Max times are set to ∞ . Also note that the markings of the Time Petri net reachability graph, discussed in the next chapter, will be equal to or a subset of the markings of the equivalent untimed Petri net. This is true since the enabling rules for the Time Petri net are the same as for the untimed Petri net. [Leveson and Stolzy 1987]

C. BUILDING THE PETRI NET MODEL

In Petri net modeling, conditions are modeled by places and events are modeled by transitions. The inputs of a transition are the preconditions and the outputs of a transition are the postconditions.

The holding of a condition is represented by a token in the place corresponding to the condition. The occurrence of an event corresponds to a firing.

The modeling and analysis presented in this thesis is concerned with the system and software at the design level and not at the implementation and software code level. Another approach or technique should be used to determine software

safety at the code level. The analysis performed in the next chapter is concerned mainly with whether the safety requirements for the design of the system are met.

1. Starting the Process

The method used to arrive at the model for the real-time system was not exactly done as proposed in this thesis. The process was started from prototype software code [Appendix A] and worked backward to create the necessary diagrams and charts. It was necessary to work backwards to a software flowchart and to piece together a few system designs to arrive at an overall system design. The situation that is recommended for the software safety analyst should be that he is presented with the flowchart and the design of the system and is then asked to perform the analysis. It is possible however to be given software and work backwards to a safety analysis of the design, using this method. If the necessary items listed below are not available, the analyst will have to create these documents to aid in building the model used for the analysis.

The main question to be answered is: after the Petri net model has been built, does the model provide sufficient information to enable software safety analysis? Then, if this model is sufficient, is there a methodology that can be used that can help to create this model. One of the crucial advantages of a methodology is that it is repeatable.

The difficult part of building the model is not to include anything in the model that was not actually specified in the design. Otherwise, this may cause the model to include a design feature that does not actually exist. This can lead to either the analysis missing a safety design problem or a design problem being added unnecessarily to the model.

The advantage of any structured methodology is that it forces the analyst to ask questions and it forces a clearer analysis at an earlier stage, where it is important to discover errors. The methodology that is proposed is presented in a step-by-step fashion, but the first model created should be the base on which other models are built. The model should be presented and discussed with the designer(s) of the system and it should be updated until all involved are in agreement that the model truly represents the design. Between the safety analyst and the designer all of these questions should be cleared up before the actual production of the system. If the analyst is unsure of a sequence of events or anything is unclear it should be cleared up while the model is being built. It is possible that during this phase some safety design problems may appear and this is beneficial because the goal of this process is for the end result to be a safer product, no matter how it is arrived at. The purpose of the model is to expose areas that need

safety design help. The benefit of any design/analysis tool is that it can help produce a safer product.

The Petri net model is created as an analysis tool, but according to Leveson [1986], since the time spent building the model is nontrivial, some of the effort may be justified by using the model for other objectives, for example, performance analysis.

In the search for a methodology, what information is needed to build the model? What types of diagrams and charts do we need to model a real-time system? To build the model it is useful to break the system into more comprehensible parts and then combine them later.

The analyst should begin by defining the system and the environment surrounding the software, then concentrate on the software. Within the system, the system elements should be defined, followed by the system flowchart being translated into Petri nets. The analyst should then concentrate on the software by translating the software flowchart into Petri nets. The last step is the combination of the system and the software by connecting all of the software/system interfaces.

As a minimum, the available information should include:

- Flowchart of software
- Flowchart of system environment
- System Elements Chart, including states of elements

- Software Elements, driven by Flowchart
- Initial Conditions (State) of the system and software
- Verbal description of each module of system
- Overall description of entire system

In order to adequately model a system all significant states of the system must be considered and explicitly presented in the model. In the translation from the design charts to the model, all significant states of the system elements must be included.

In general, if in building the model some extra information about states of the system were included in the model, it would not affect the model, but might add to the clarity, while adding some complexity. Adding unnecessary items to the model is not optimal, but if during construction of the model the analyst is unsure, he should include the information and not throw it away; it might be needed later. The experience of the analyst would help to decide which states would be important in terms of overall system safety.

The key is to remember that the Petri net model tries to include all possible states that the system can reach. The analysis of the model concentrates on whether the system either cannot reach hazardous states or that if it does, the system can control the result.

2. Defining and Modeling the System Elements

The best place to start the modeling is with the system surrounding the software. Before looking at the overall system flowchart it is necessary to first define the elements of the system. To make a list of these, the analyst needs to review the system flowchart and the system descriptions. A complete listing of all the system elements is included in Appendix D and the corresponding Petri net model in Appendix E.

To give some examples of the system elements and how they are modeled, a few are examined below. The following is a detonation interruptor element, which moves a third of the way towards removal each time the solenoid toggles. Thus, after three toggles of the solenoid and the unlocking of the safety arming device, the detonation interruptor is removed. Figures 4-13 and 4-14 show how this is described in terms of system elements.

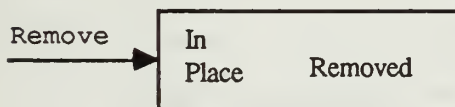


Figure 4-13. Detonation Interruptor

Figure 4-15 below shows a direct translation from the system elements to the actual Petri net model. The model shows the necessary inputs to remove the detonation

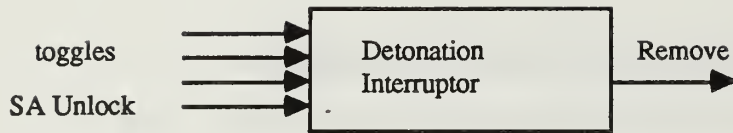


Figure 4-14. Detonation Interruptor Removal

interruptor and the resulting output is the removal of the interruptor.

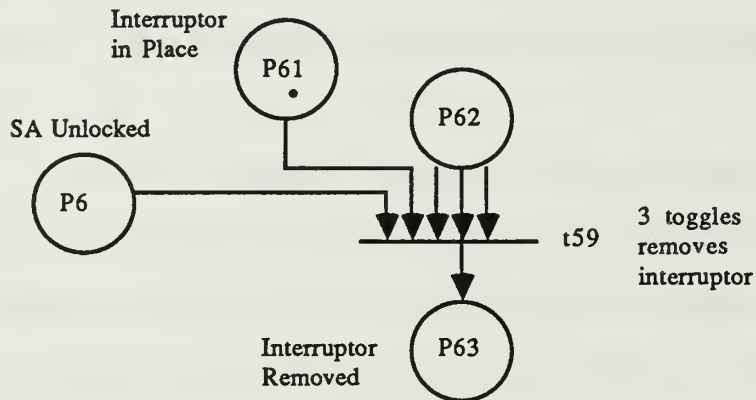


Figure 4-15. Petri net model of detonation interruptor removal

In Petri net modeling there are many instances of binary or two-state occurrences, i.e. on/off, left/right, or enable/disable. In Petri net modeling there are two ways to describe these. The differences depend on whether the interface with the two-state elements requires continuation feedback or whether the state is changed and the system moves on. For example in Figure 4-16, another system element, the Analog/Digital Converter is described.



Figure 4-16. Analog/Digital Converter

Figure 4-17 shows the translation of the Analog/Digital Converter into a Petri net model. In the system model this element requires continuation feedback to the next step of the model. After the enable transition, the next transition in the system to input data from the analog/digital converter. Thus, the first transition is necessary to allow the next transition.

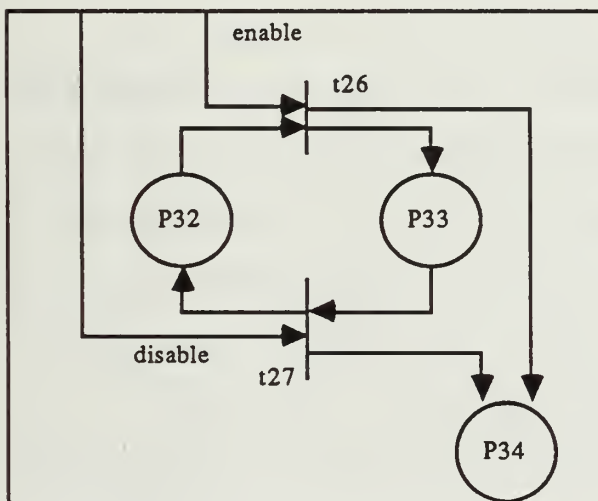


Figure 4-17. Analog/Digital Converter Petri net model

The Intent to Launch (ITL) Sensor, shown in Figure 4-18, is an example of the other type of two-state element

with no feedback. Initially, the ITL Sensor is off meaning that the ITL signal has not yet occurred. When the ITL signal is received the transition causes the two-state element to switch to on.

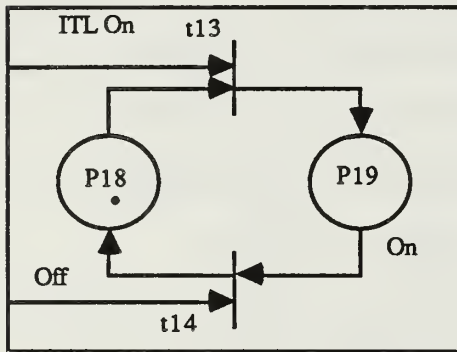


Figure 4-18. ITL Sensor Petri net model

To provide a methodology for building a Petri net model it is useful to provide common abstractions that can always be modeled the same way. Such abstractions are the two types of binary state system elements. Figure 4-19 shows the generic model for a two-state element with feedback and Figure 4-20 shows a two-state element without feedback. These are exact duplicates of Figures 4-17 and 4-18 (from the real system), where the common abstraction allows two transitions, i.e. turn on and turn off.

One typical difficulty with creating this type of diagram (model) is that once the model is built and put on paper no one besides the builder can understand or read it,

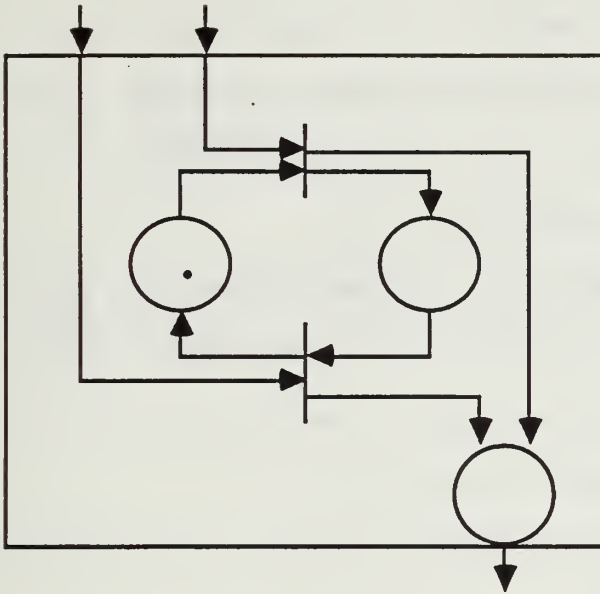


Figure 4-19. Two-state element with feedback

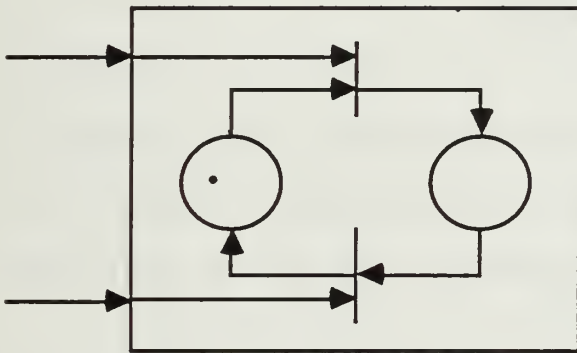


Figure 4-20. Two-state element without feedback

because it is far too complex, with lines going all over the place. In order to simplify the model, the system elements can be represented as black boxes in the model and the inside states can be represented on a separate chart. Also to simplify the model if a system element is accessed more than

once in different areas of the model, for readability it is useful to repeat the element and signify that it is the same element with a certain marking. In this model repeated elements have the same border markings.

The few items that have just been mentioned are an attempt to manage complexity problems. If the design is to be changed or updated and then the model, there must be a systematic discipline to manage the updating.

Once all of the system elements have been defined and their corresponding model has been built, the focus of attention can move to the actual translation of the system flowchart, including, as the work progresses, the system elements as they fit into the model.

3. System Flowchart

A system flowchart of the events occurring in the system should have been prepared by the designer of the system. A complete system flowchart for the guided missile system is included in Appendix C.

There is a direct translation from the system flowchart to the Petri net model. Figure 4-21 illustrates the translation from the system flowchart to the Petri net model, which will make this statement more apparent.

During this translation, the system elements, in this example the state of the computer, which interface with the system flowchart should be added. It is difficult to decide

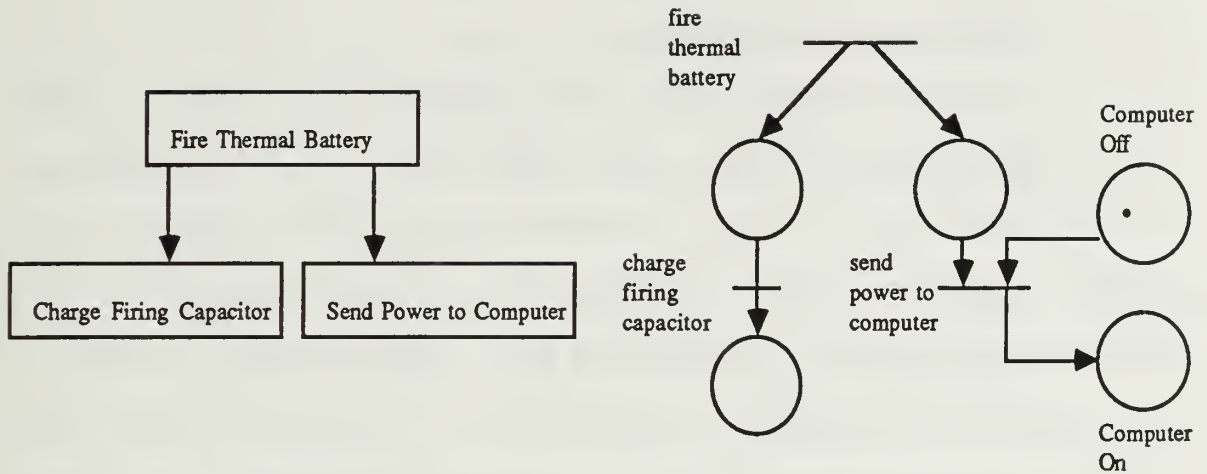


Figure 4-21. System Flowchart translation to Petri net

whether all binary states (on/off) of the system elements need to be explicitly mentioned to determine software safety, but it is better to include them in the model for a more complete representation of the state of the system. The additional states will not affect the model and might actually add more clarity. Unfortunately, it also adds more complexity to the model, but do not worry about that at this point.

At this point, the reader has probably realized that it is difficult to separate the various parts of this process and there is definite overlap between parts of the processes. If the proposed methodology is followed all of the components of the system should be included. The main concern is to model all states of the system.

4. Modeling the Software

With the system level now modeled the analyst can move into the software modeling. The first step is to translate the flowchart directly into a Petri net and then add the software elements which interface with the software. Then the system level should be interfaced with the software, making sure to connect the system components. This should actually be done simultaneously with the software flowchart translation.

Petri nets represent the control structure of programs well. Since a flowchart represents the flow of control in a program, it is very similar in nature to a Petri net.

[Peterson 1981]

A flowchart is composed of nodes of two types: decisions represented by diamond shapes and computations represented by rectangles, and arcs between them. Petri nets model the sequencing of instructions and the flow of information and computation but not the actual computation values themselves. The Petri net of the software is an abstraction of the modeled system. The appropriate translation from a flowchart to a Petri net replaces the nodes of the flowchart with transitions in the Petri net and the arcs of the flowchart with places in the Petri net [Peterson 1981]. Each arc of the flowchart is represented by exactly one place in the corresponding net. A convenient way to execute a

flowchart is to introduce a token which represents the current instruction. The flowchart, in this sense is actually representing the state of the software in the form of a state-transition diagram.

Figures 4-22,4-24, and 4-26 show some generic examples of how flowchart symbols can be translated into Petri net modeling symbols. In a software flowchart the main two events are computations and decisions.

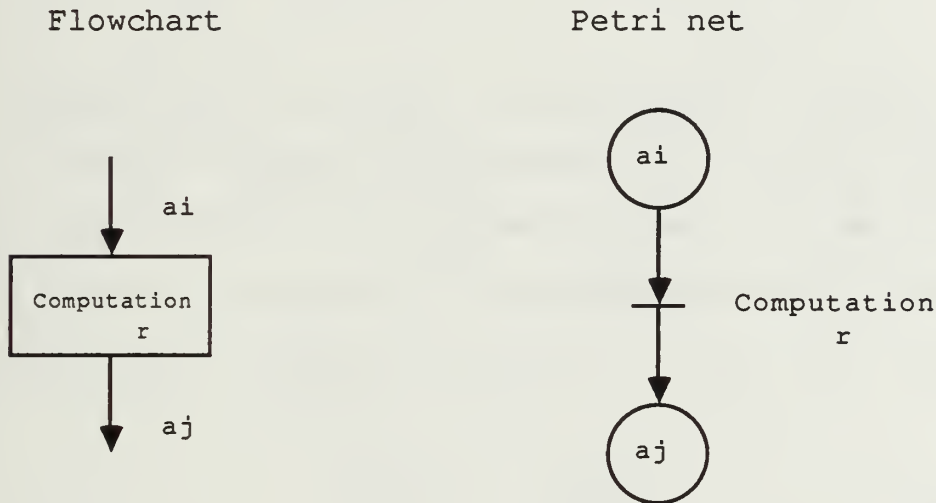


Figure 4-22. Petri net model of a flowchart computation

For the complete software flowchart, the reader is referred to Appendix B. Figure 4-23 is an example from the real-time system, illustrating the concept shown in Figure 4-22.

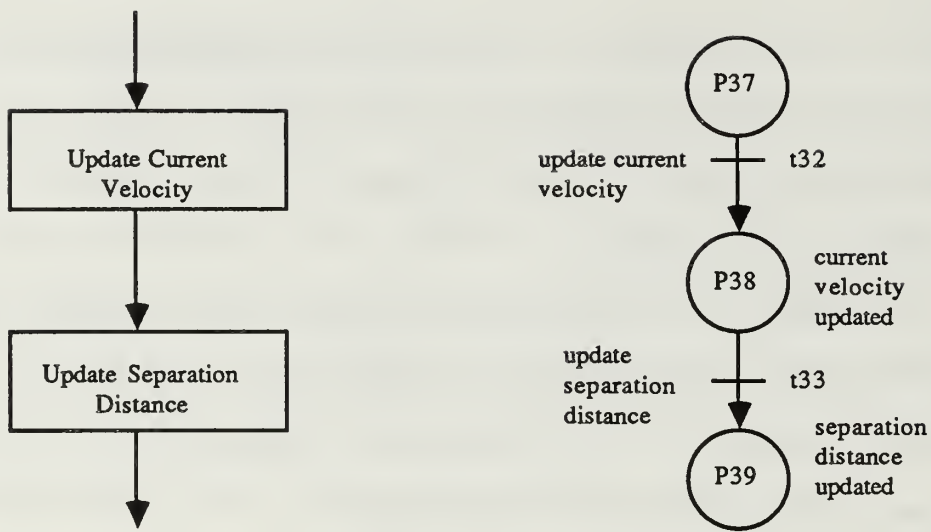


Figure 4-23. Flowchart translation to Petri net model

In order to model a decision there are two possible interpretations. The first interpretation assumes that an external agent just resolves the decision and goes directly to the next state. This interpretation is displayed in Figure 4-24.

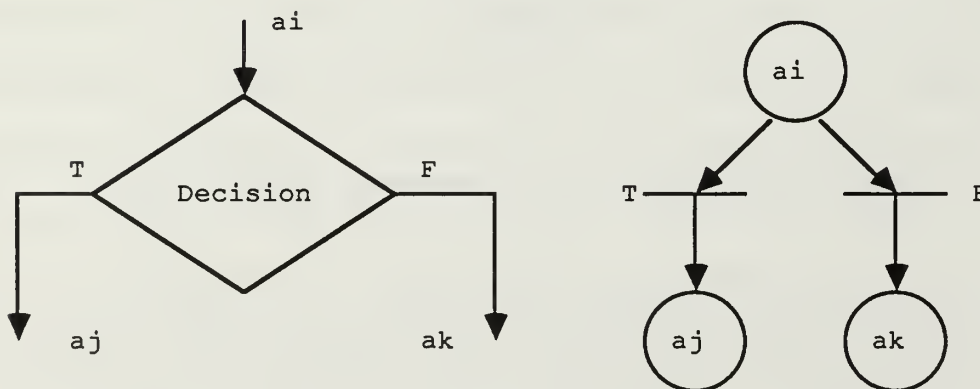


Figure 4-24. Modeling a software decision

Figure 4-25 displays the interpretation used for this thesis. This interpretation takes into account that there is an additional event, which is an actual check or comparison in order to make the decision, but that the decision is made by an external agent. Figure 4-26 is a real-time system example of the second interpretation.

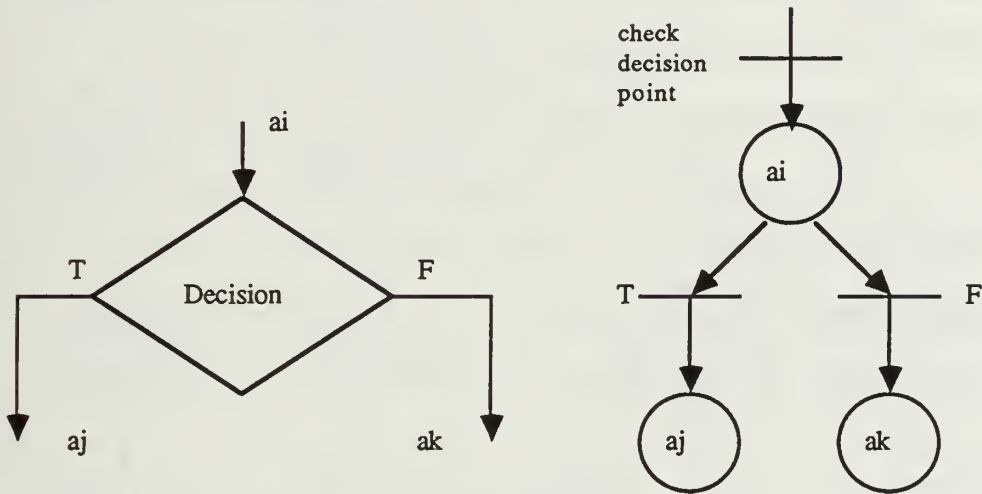


Figure 4-25. Software decision as a Petri net model

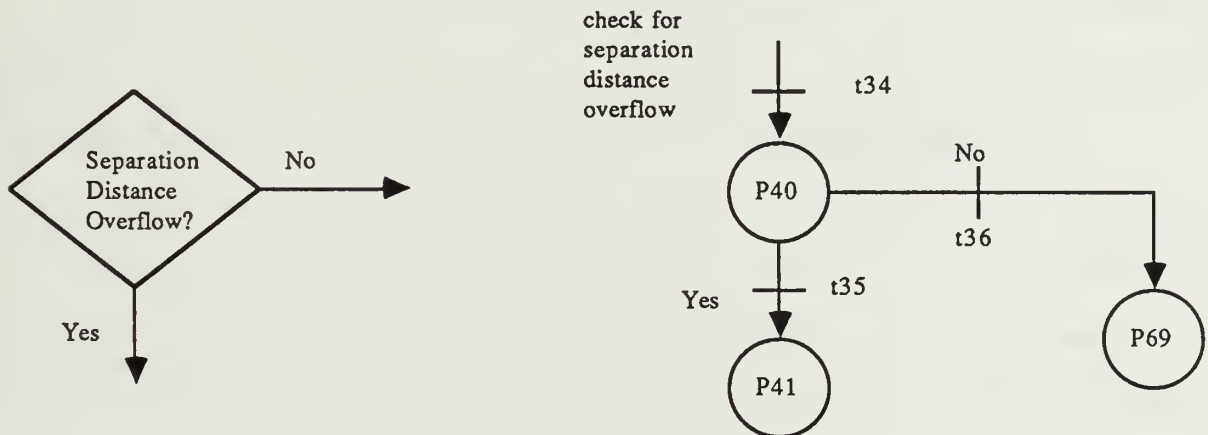


Figure 4-26. System Example of Flowchart to Petri net model

An OR condition is equivalent to a decision and should be modeled the same way. The model consists of one place with two output arcs from it, leading to the two transition choices, allowing an external agent to choose one path or the other.

While reviewing the software flowchart all software elements such as counters or any binary variables, that affect a state should be included. Counters can be modeled as a single state and the number of tokens present in the state can model the count. The counter of the state of the pointer is a software element in this system. This pointer variable counter can be in one of three states. As the pointer value is increased the state is represented by the number of tokens present in the place.

Figure 4-27 is a larger example of a translation from the flowchart to the Petri net model. The similarities between the two are readily visible when they are presented side by side.

The system elements included in this section of the flowchart and model are the A/D Converter, Figure 4-17 and the ITL Sensor, Figure 4-18. In this excerpt, the A/D Converter is repeated in four different places. This is the same system element and this is represented by the identical border around the element name. Inside this element are the same three states and two transitions. The enable and disable lines

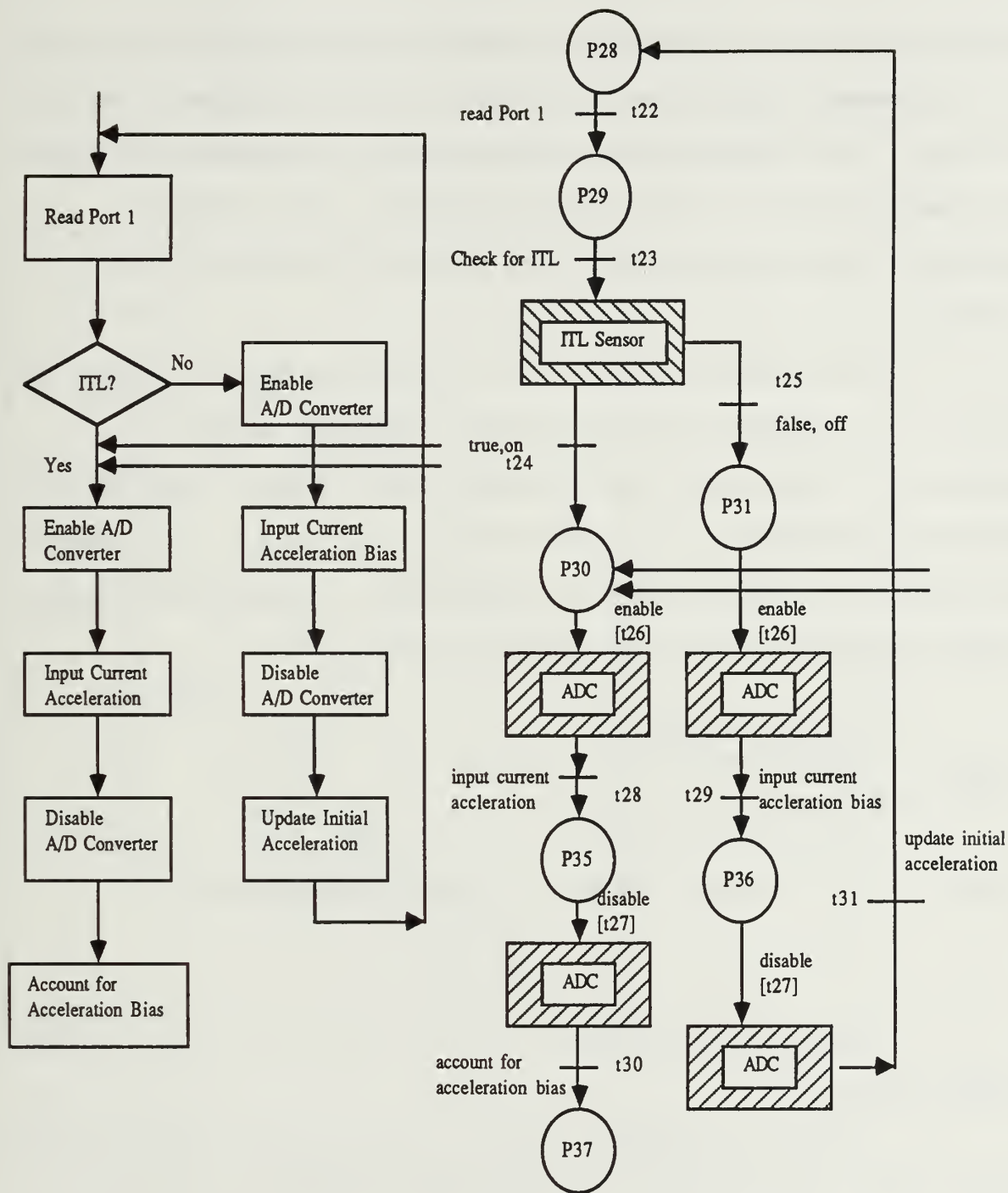


Figure 4-27. Translation from Flowchart to Petri net model

leading to the transitions t26, and t27 respectively (denoted by the number in brackets), describe where the input line is leading. The feedback line coming out of the element is from the feedback place in the system element. The reader is referred back to the appropriate figures to make this point clearer.

In software modeling, as in system modeling, there is the need to abstract common elements that can always be modeled the same way. The software elements that lead to this modeling are shown in Figures 4-28 and 4-29 and are followed by Figure 4-30 which shows an example of the real-time system toggling mechanism for the solenoid status.



Figure 4-28. Software Element for Toggling Mechanism

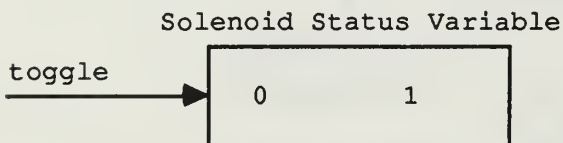


Figure 4-29. Description of two-state Variable

When the two system elements are put together, the end result is the toggling mechanism for the solenoid status.

Without the specific place numbers and transition numbers, this can represent a generic abstraction for any toggling mechanism including a binary state variable.

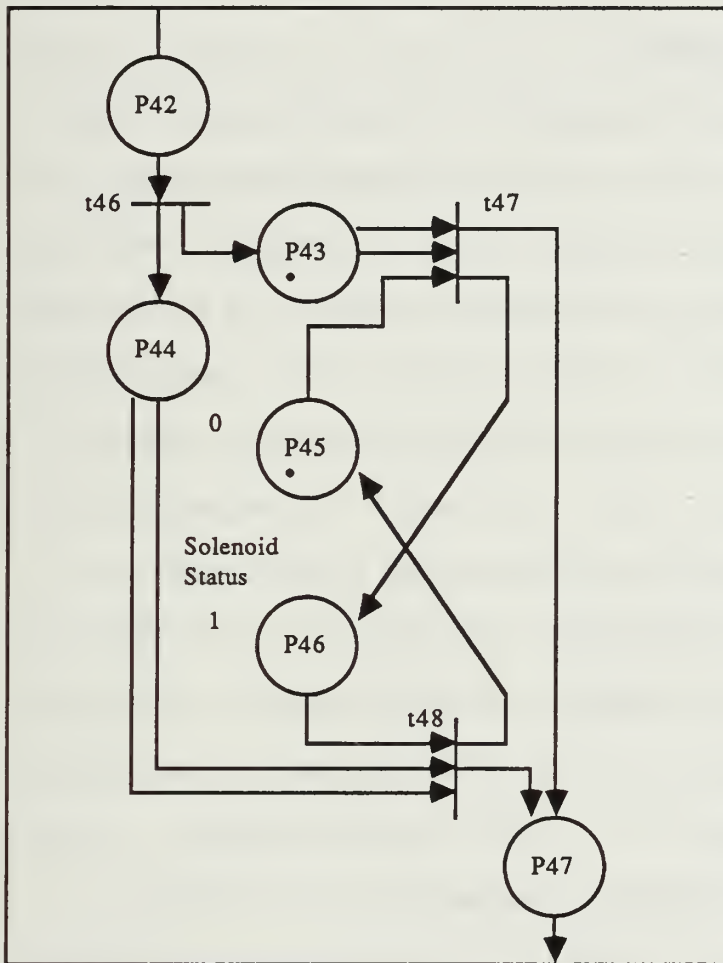


Figure 4-30. Toggling the Solenoid Status

5. Completing the Model

Now that the analyst has put all of the parts of the model together, it is time to review the model for

completeness and to present the model to the designer(s) of the system. To construct a final model of this system it took three different versions and two reviews of the model.

D. DIFFICULTIES WITH MODELING

The major difficulty with modeling in general, not just with Petri nets is, what level of detail does the analyst model in the system? As the analyst goes through the building process what are the subjective judgements that are made about the system? A methodology has been proposed that constructs a model, but it is difficult to say that this is the optimum model available given the system. The model represents an abstraction and when the analyst is dealing with system elements should the model represent an abstraction of the element or model the exact function of the element. The analyst needs to pick a happy median and hopefully this thesis provides the steps necessary to arrive at that median.

When modeling system elements and decisions on these elements the inner workings or details of the system are not necessarily present. Except for gross errors in building the model, a system designer can probably always correct the model saying that the system might not exactly work the way it is modeled. The key to remember is that this model is just an analysis/design tool for the system and is not supposed to actually be the system. The more realistic the model is

though the more it will tell the safety analyst about the safety of the system.

If the model does not provide exact duplication of the system, the model allows safety points in the system to be readily apparent. In the case of decisions, there is a choice to show a decision resolved by an external agent or to model the system exactly as it would occur without giving the model a decision (similar to hardwiring the system). Leaving the decision as being determined by an external agent allows the decision point to be viewed as a critical point in the system. All decisions points are usually also safety point considerations.

For the analyst the choices are not always clear-cut as to how to model a certain part of the system. It is useful to consider choices at an early stage of the modeling process. At the end of the process one choice might become more apparent to the analyst.

In proposing a methodology it is difficult to remove the subjectivity portion of the modeling experience. Three different analysts might construct three slightly different models. This chapter has provided a methodology to follow which discusses the elements of Petri net modeling and the issues concerned with safety modeling. It is unrealistic for the analyst to believe that by following this methodology and performing the subsequent analysis, discussed in the next

chapter, that this will provide a perfectly safe system. Petri net modeling is only a design/analysis tool to help construct a safer system.

E. SUMMARY OF MODELING METHODOLOGY

In this chapter, a methodology has been proposed to construct a Petri net model for software safety analysis. Petri nets are able to model the hardware and software of a system all in one net. The analysis performed in the next chapter will tell whether or not this model provides adequate information for safety analysis.

V. ANALYSIS METHODOLOGY AND RESULTS

A. INTRODUCTION

The purpose of this chapter is to examine the concepts of software safety analysis and determine if the Petri net model constructed in the previous chapter is appropriate for safety analysis. This chapter does not attempt to present a complete analysis of the system, but instead examine a representative number of problems to explain the technique and present a conclusion on the appropriateness of the model. The analysis technique presented and discussed in this chapter is due to the work of Leveson and Stolzy. For more information on this technique see [Leveson and Stolzy 1987].

The purpose of system safety modeling and analysis is to show that the system is safe both if it operates as intended and in the presence of faults. This analysis attempts to detect hazards in the system design and discusses possible design modifications to eliminate them.

The primary function of the software in this system is to monitor the surrounding system and perform the safe separation distance calculation.

The overall goal in designing a safety-critical system is to eliminate hazards from the design or to minimize risk by altering the design so there is a very low probability of the

hazard occurring. According to Leveson and Stolzy [1987] in "most realistic systems it is impossible to completely eliminate risk. The goal instead is to design a system with acceptable risk." To show that a system is low-risk or safe, it is necessary to first ensure that if the specifications are correctly implemented and no failures occur, operation of the system will not result in a mishap. The first part of the analysis will focus on possible hazards the system can achieve under normal operating conditions. Second, the risk of faults or failures leading to a mishap must be eliminated or minimized by using fault-tolerance or fail-safe procedures. If it is not possible to completely eliminate the possibility of a hazard occurring, then in order to reduce risk the length of time of occurrence of the hazardous conditions must be minimized.[Leveson and Stolzy 1987]

In general, from a safety standpoint, the first priority of the response to a safety-critical situation is the reduction of risk rather than attainment of mission.

The analysis performed in this thesis does not consider the correctness of the algorithm or the correctness of the software or system design. This analysis is concerned with the relationship between the software subsystem and the overall system.

B. ANALYSIS METHODOLOGY

1. Preliminary Hazard Analysis

The first step in any safety analysis is to identify hazards and categorize them with respect to criticality and probability (i.e. risk); this is called a preliminary hazard analysis (PHA). Potential hazards that need to be considered include normal operating modes, maintenance modes, failures or unusual accidents in the environment, and errors in human performance. [Leveson 1986]

Hazards can be categorized by the aggregate probability of the occurrence of the individual conditions which make up a hazard and by the seriousness of the resulting mishap. A mishap is an unplanned event or series of events that results in death, injury, illness, or damage to or loss of property or equipment. Together these constitute risk. [Leveson 1986]

A beneficial source of information on preliminary hazards should be the system designer. The system designer should be aware of the system safety issues, in addition to the standards that this system must meet.

Since for safety-critical systems, safety is the primary concern, the benefit of beginning this analysis as early as possible in the design phase is important. A safety analyst should be included in the preliminary design phases to simplify the analysis process. The analyst could make an

impact on the design of the system earlier in the process and help save time and money.

Once hazards are identified, they are assigned a severity and probability. Often early in the design of the system the probabilities are unknown and the analysis is done considering only the severity. Classification of hazard severity is important, as to low-risk, medium-risk, or high risk. [Leveson and Stolzy 1987]

The safety features for the design of a safety arming device are contained in [MIL-STD-1316C]. There must be two independent safety features: 1) to prevent unintentional arming and 2) to provide forces to remove safety features derived from different environments. Operation of at least one of these safety features shall depend on sensing a post-launch environment.

Safety features shall provide arming delay. Arming delay shall provide safe separation distance for all defined operational conditions. Also, an assembled fuze shall not be capable of being armed manually.

From the PHA of the guided missile fuze safety arming device a hazard exists:

- 1) if the weapon is assembled and the firing capacitor is charged, since there is a high voltage present at the safety arming device interface.

- 2) if the firing capacitor is charged, there is the potential for inadvertent, premature detonation, before the safe separation distance is reached.

In general, the primary safety concern for this system is premature or inadvertent detonation. To show the concepts of the analysis technique, the analysis will examine several representative samples of sequences of events and/or failures that can result in premature or inadvertent detonation.

At this stage, the designer and analyst should discuss all possible failures to the system that are known. The analyst may discover more during the analysis, but if this is discussed now it will save time when failures are inserted into the model.

Three possible high-risk states are analyzed in this section:

- 1) Can the missile detonate without receiving a 4+G Boost (failure condition)?

In the Petri net model, the state of the system is represented by a collection of places each with a token. The places are P15 (rocket motor fired, 4+G Boost not received), P67 (signal to fire received), and P17 (firing capacitor charged), and P14 (missile rack released).

- 2) Can the system reach the high-risk state of P18 (ITL signal not recieved), P67(signal to fire received), and P17(firing capacitor charged)?

Additional problems considered, but not examined in this thesis are:

- 1) Since the thermal battery can fire from electrostatic discharge and accidentally start the computer and software, can this lead to premature detonation?
- 2) Can the system accidentally toggle the solenoid three times before we have reached the safe separation distance, allowing premature detonation?
- 2a) Can a problem occur if the value in the safe separation distance lookup table is either incorrect (possibly data destroyed) such that when the check for safe separation distance in P70 occurs the result is incorrect. This could lead to no detonation of the missile, since the detonation interruptor would never be removed?
- 3) Can the voltage to the solenoid be applied too long and burn out the solenoid? The timer loop is currently 200 ticks which is five times the minimum time needed to toggle the solenoid.

2. Software Hazard Analysis

Once the PHA is completed, software hazard analysis can begin. Using the hazardous states which have been identified

in the PHA, it may be possible to work backward to the software interface using Petri net analysis techniques such as those described by Leveson and Stolzy [1987] to derive the software safety requirements.

In this research, the Time Petri nets are not used, but the traditional concept of Petri nets is used. There are certain parts of the analysis that can be performed on untimed or traditional Petri nets. In the methodology proposed, it is natural to first build the untimed Petri net model, so this analysis will focus on that first. The analyst can show that certain states are reachable without considering timing constraints, but when timing constraints are added, the state might be unreachable. The addition of timing constraints is important for safety considerations which are affected by the timing sequence of events. In real-time systems, correct software actions which are too early or too late can lead to unsafe conditions. [Leveson and Stolzy 1987] In this system any analysis concerned with whether or not the system has reached a safe separation distance is concerned with timing constraints.

a. Building the Reachability Graph

To show that a system is low-risk or safe, it is necessary to first ensure that if the specifications are correctly implemented and no failures occur, the operation of the system will not result in a mishap. In order for the

analyst to determine if the system design can "reach" any high-risk states, the analyst creates a reachability graph. [Leveson and Stolzy 1987]

The reachability graph identifies all possible states that the system can reach from the initial state by any legal sequence of transition firings.

Given an initial state, the reachability set for the Petri net is the set of states that results from executing the Petri net. This graph will help the analyst to eliminate high-risk hazards which have been designed into the system.

In this thesis, a reachability graph is created where the nodes of the graph are labeled with the present marking (i.e. the state) and the arcs represent transitions between the states. For further information, the theory of reachability sets is presented in Appendix I.

By using the inverse Petri net, where the input and output functions are reversed, it can be determined if a high risk state is reachable by using the high risk state as the initial state and determining whether the original state is reachable.

The analysis performed below is to check if under normal operating conditions, the high risk state of P18 (ITL signal not received), P67 (signal to fire received), and P17 (firing capacitor charged) can ever be reached.

The reachability graph begins with the high-risk state which consists of tokens in P18, P17, and P67. The graph is built by firing transitions while working backwards. In Figure 5-1 places which do not provide any useful risk information are dropped from the graph (i.e. P66 and P61). The notation keep P62(2) represents two tokens remaining in P62, but since this is a large loop, the tokens are consumed in the same manner through each loop and this is not repeated in this graph.

The backwards analysis graph depicted in Figure 5-1, shows that there is no path to the initial state from this high risk state, so this cannot occur under normal operating conditions. The ITL signal check by the software inhibits this state from ever occurring under normal operation.

This analysis works well, but generating the entire reachability graph is a very time consuming task and the graph is large for a system of even this size. It is possible for the backward reachability graph to be as large as or even larger than the original graph. The use of an automated tool to create this graph is almost a necessity for any real-time system and is definitely the case for this system.

Leveson and Stolzy [1987] propose a different technique which allows the analysis of the design without

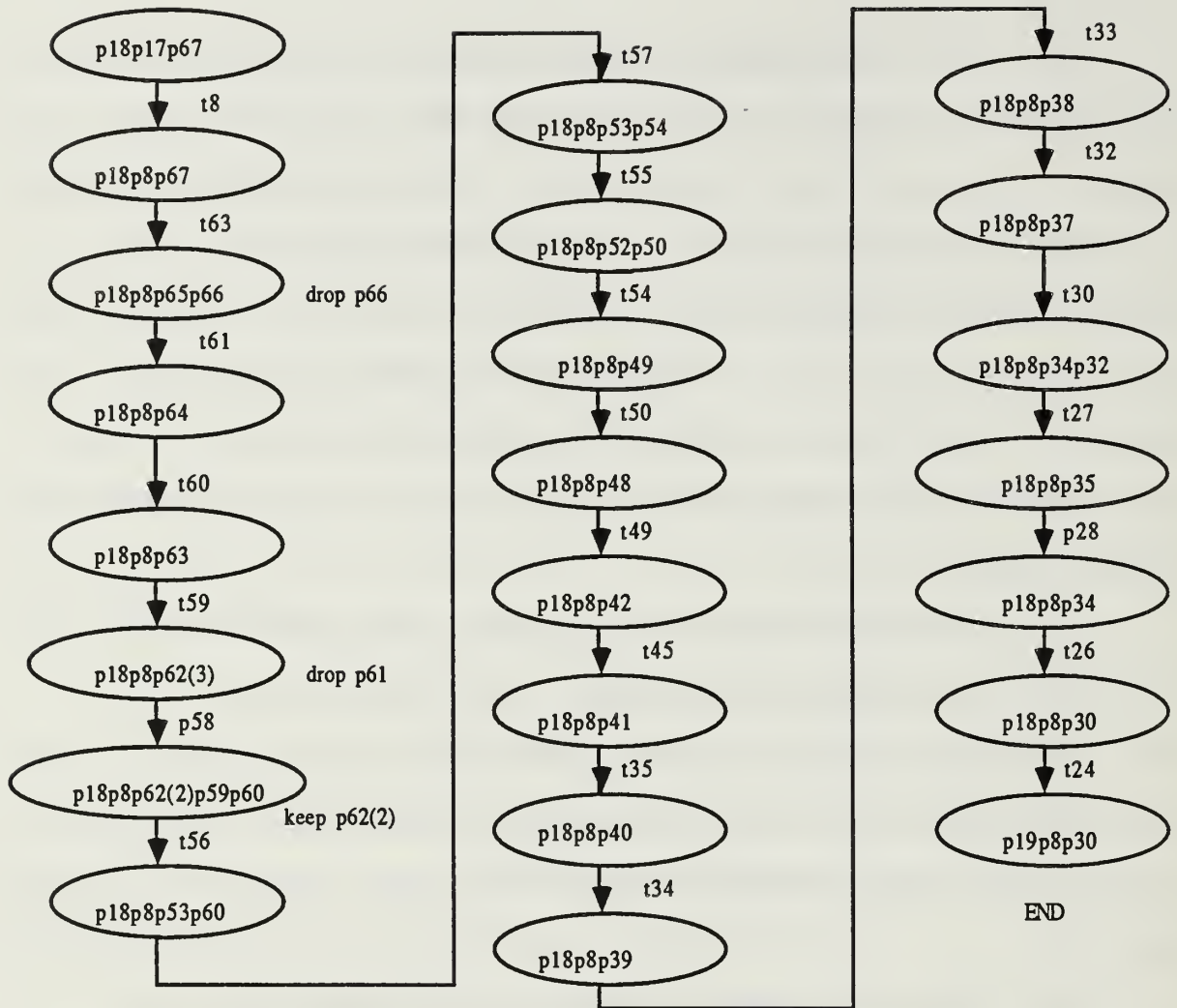


Figure 5-1. Backwards Reachability Graph of Detonation without Intent to Launch

creating the entire reachability graph. Their approach is to identify and eliminate high risk hazards that have been designed into the system. Their technique makes the high risk states unreachable, by eliminating the critical path to the high risk state. This algorithm does not require construction of the entire reachability graph, but requires the definition of critical states.

Using the Leveson and Stolzy technique, the states of a reachability set are separated into two disjoint sets: states from which it is possible to reach high-risk and possibly also low-risk states and those from which it is possible to reach only low risk states.

definition: A state (marking) μ_C is a critical state if and only if:

- a) $\mu_C \in$ low-risk states and
- b) there exist two nonempty sequences of transitions s_1 and s_2 and two markings μ_i and μ_j such that $\delta^*(\mu_C, s_1) = \mu_i$ and $\delta^*(\mu_C, s_2) = \mu_j$, where $\mu_i \in$ high-risk states and $\mu_j \in$ low-risk states.

If a high risk state is reachable, then there must be a critical state on the path from the initial state to the high risk state (this includes the possibility that the critical state is the initial state). The approach is thus to define all the hazardous states of the system and eliminate the path to them, by creating an alternate low-risk path.

To ensure that high-risk states can never be reached, it is possible simply to work backward to the first critical state (i.e. to a state in the reachability graph that has two successors) and to use design techniques such as interlocks to ensure that the high-risk path is never taken.

A critical state is a state from which a low-risk state is reachable by a transition and a high-risk state is reachable by a transition. To eliminate the critical path which proceeds through this state, it is necessary to guarantee the system takes the low-risk path and not the high-risk path. The high-risk path can either immediately or potentially lead to a hazardous state.

"Using the backward technique to determine if a high risk state is reachable is useful when the goal of the analysis is to prove only that the system cannot reach certain hazardous states. The backward approach is practical only if one considers a relatively small number of high risk states." [Leveson and Stolzy 1987] In most real time systems, however, the number of truly high risk states should be a manageable level. If this is not the case, then it is worthwhile to produce the complete reachability graph.

The disadvantage of this technique is that it is conservative, in that the analyst may define a larger number of critical states than are actually needed. To reduce the large amount of computing to produce the entire reachability graph, critical paths are defined, which might not have been defined if the entire reachability graph for each hazardous state was created. [Leveson and Stolzy 1987] Since the entire graph is not created, this approach is a short cut, but at the same time could lead to more design changes than might be necessary.

According to Leveson and Stolzy [1987], "eliminating a nonexistent path may have the effect of eliminating or lessening the possibility of mishaps caused by run-time faults and failures." Using the algorithm, if an uneliminated path also leads to a mishap, this will be determined in a later step and the this path will also be eliminated.

The complete algorithm of Leveson and Stolzy [1987] is in Appendix J. The algorithm starts with the set of high-risk conditions. "For each member of this set, the immediately prior state or states are generated. Each of these "one-step-backward" states is then examined to see if it is a potentially critical state and can be used to eliminate one path to the high risk state. Note that the algorithm starts with partial states and not complete states. That is, some conditions in the state are unimportant as far as risk is concerned, and thus it is not known at the beginning of the algorithm the complete composition of the reachable high-risk states (the complete states from which to start the backward analysis). The don't care places in each state are "filled in" with those conditions that are possible in the process of executing the algorithm. Finally, the analyst only needs to look forward one step from each potentially critical state in order to label it as critical (i.e. there exists a next-state that is low risk). This is because if this path also leads to a high risk state, then it will be eliminated by the algorithm in a later step."

To show a comparison between the two methods, the first problem from the list is examined. Can premature detonation occur with missile released from the rack and without the occurrence of a 4+G Boost.

Under normal operating conditions this could not occur, because of the timing constraints, but due to a failure, the analysis will show if it is reachable. Even though this net is untimed, the time for the 4+G Boost is definitely less than the time needed to calculate the safe separation distance under normal operations.

Before the problem is analyzed, the concept of inserting failures must be discussed. In the Leveson and Stolzy method a fault is represented by a failure transition, which acts like other transitions but is denoted by a double bar and a fault condition which is denoted by a double circle, shown in Figure 5-2.

In this thesis we are concerned only with determination of which failures can potentially lead to a hazardous state, that was unreachable without failure. Leveson and Stolzy [1987] propose methods for making a system fail-safe or fault-tolerant. A further discussion of failures inserted into the system is discussed in section D of this chapter.

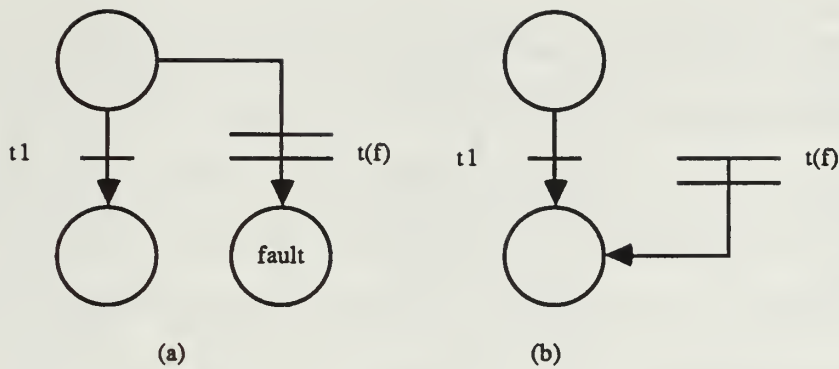


Figure 5-2. a) Desired event $t1$ doesn't occur; b) Undesired event $t1$ occurs

The high risk state is P15 (rocket motor just fired, but no 4+G Boost), P17 (firing capacitor charged), P67 (signal to fire recieved), P12 (missile on the rack). Figure 5-3 shows the insertion of a failure into the net. This failure causes the removal of the token from this place, causing the next transition not to occur (4+ G Boost). The failure represents the rocket motor not firing or if the rocket motor fired it did not reach the necessary 4+G Boost.

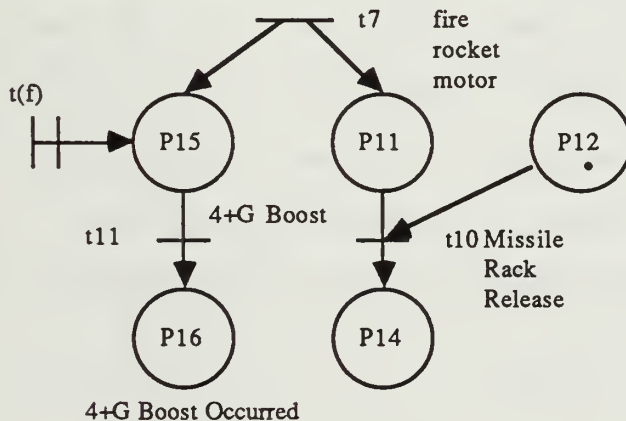


Figure 5-3. 4+G Boost failure in place 15

The backwards reachability graph for this high risk state was constructed and is shown in Figure 5-4. The graph, starts from the hazardous state and works backward to show that the initial state P0 is reachable if this failure occurs.

Figure 5-5 shows the critical state definition method as proposed by Leveson and Stolzy [1987]. Notice that two critical states have been defined. In the next section a method will be shown that eliminates this bad (critical) path from the design. The critical transition is t64 (detonation) which must occur prior to t11 for the system to work normally.

C. SYSTEM DESIGN MODIFICATIONS

The modification methods discussed in this section modify the current model and subsequently the design. The model is just a tool to show the result and aid in the analysis. The decision as to whether the modification should be made in hardware or software should be made by the designer.

To eliminate the high-risk path from the design in the previous example, it is necessary to modify the Petri net in some way to ensure that the safe path is always taken, i.e., that another transition is always performed before or has precedence over the critical transition. [Leveson and Stolzy 1987]

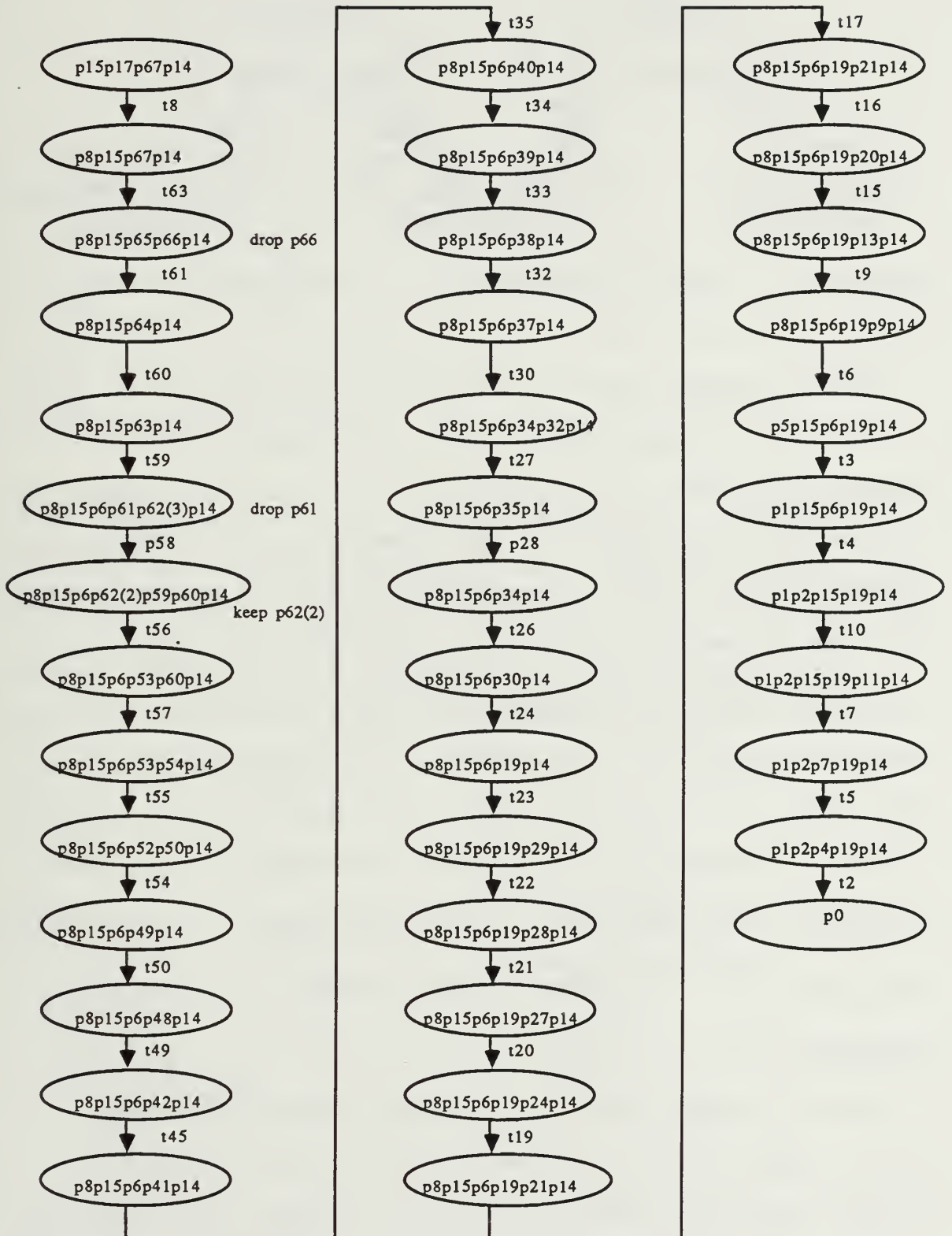


Figure 5-4. Backwards Reachability Graph for 4+G Boost Failure

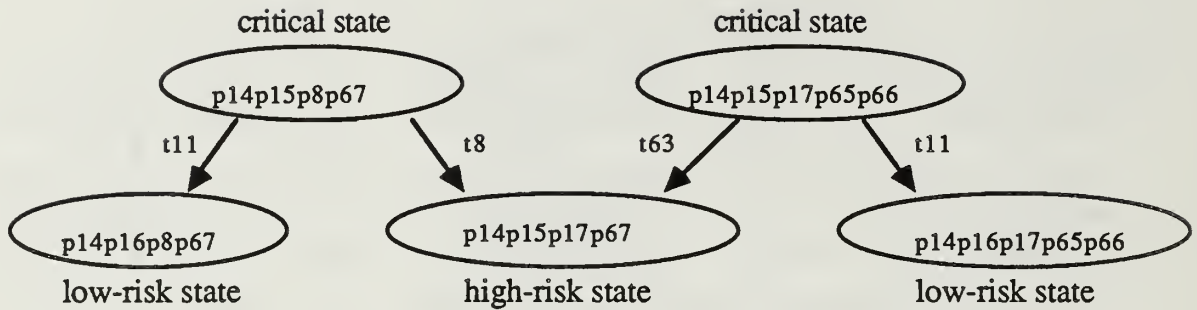


Figure 5-5. Critical State Method for 4+G Boost Problem

There are many ways to modify the system design in order to eliminate the high-risks states. This may lead to permanent patches of the design or it may require a redesign of a section of the system. One common approach is to use an interlock, Figure 5-6. Interlocks are used to ensure correct sequences of events. To model an interlock in a Petri net, assume that t_2 is the desired transition, while t_1 is the undesired transition. It is possible to force the system to always take the desired path by making the following changes to the two transitions in the net. The approach is to add a new place (the interlock I) to the output bag of t_2 and to the input bag of t_1 . This ensures that transition t_2 always has precedence over t_1 .

If an interlock were used in this example it would connect transition t_{11} (4+G Boost) to t_{64} (Detonation), thereby requiring a 4+G Boost to occur, before the detonation

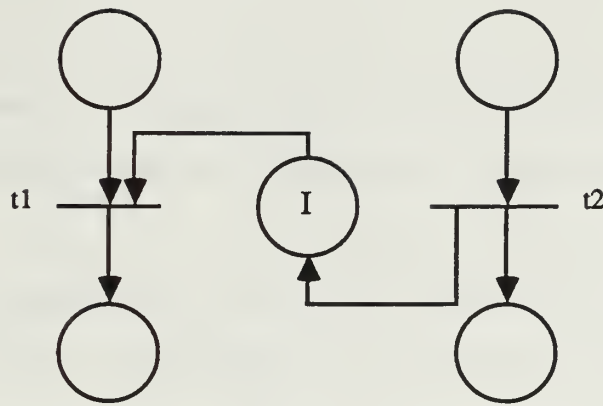


Figure 5-6. Interlock

of the missile. This is shown in Figure 5-7. This design modification could be implemented as a 4+G Boost check in the software somewhere after the check for Intent to Launch. To provide a system that is fail-safe or fault-tolerant the system designer will have to decide what mechanism to add to

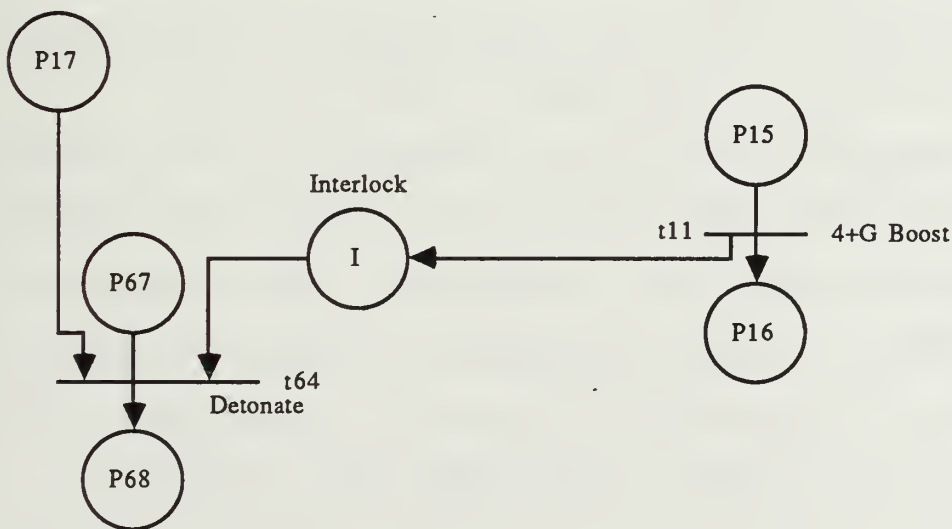


Figure 5-7. Interlock to modify 4+G Boost design

provide fault-tolerance or a fail-safe system. [Leveson and Stolzy 1987] have proposed a technique for providing a fault-tolerant or fail-safe system.

Since premature or unsafe detonation is the primary concern of this system, most of the interlocks will connect with the critical transition t64 thus attempting to prevent premature detonation.

In this example, creating the backwards reachability graph showed that with a failure the hazardous state in question was reachable. By performing the algorithm proposed by Leveson and Stolzy this high-risk path could have been realized and avoided sooner.

Another type of locking mechanism, Figure 5-8, ensures that an event does not occur while another condition is true. This is implemented in the Petri net by using a locking place. With a token initially in the place L (lock) only one transition (either t1 or t2) can be enabled. For this example say t1 is enabled and fires to place the token in P2, enabling t3. Only after t3 fires can t2 be enabled, thus allowing t1 or t2 to be enabled. This corresponds to a critical section in software, which is also shown in the mutual exclusion net shown in Figure 4-2. [Leveson and Stolzy 1987]

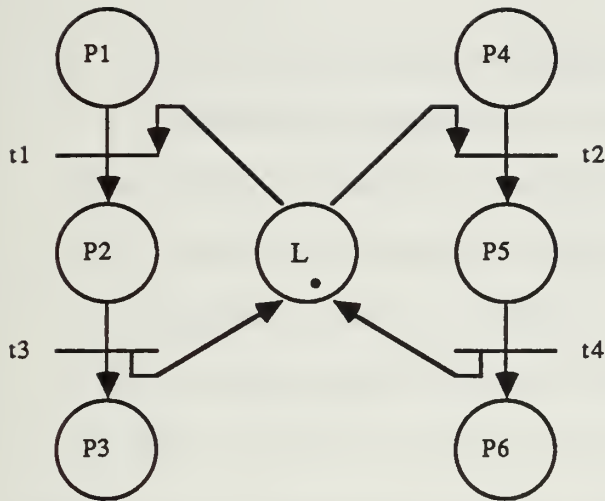


Figure 5-8. Locking Mechanism

Another way to ensure that one transition will always fire when both are enabled is to enforce timing constraints or timing conditions in the designed system. To ensure that t_j does not fire when t_i and t_j are both enabled, the following timing constraint must be enforced: the maximum time that it may take for the desired transition (t_i) to fire must be less than the minimum time for the other transition (t_j) to become enabled and to fire. Each of these time quantities must be the total time that the enabling conditions have been met, not just the individual transition time limit. [Leveson and Stolzy 1987]

One method to determine these times is to use the reachability graph to find the maximum (or minimum) valued path leading to the transition that is continually enabled.

Timing constraints are enforced in systems by either verifying that the design makes it impossible for the constraint to be violated or by using a watchdog timers and other devices to determine when the constraint is about to fail and to insert recovery techniques (either hardware or software) into the system design. [Leveson and Stolzy 1987]

These procedures only identify possible ways to modify the design to make it safer. The actual interlocks and timers that are used must be considered from an engineering feasibility and cost standpoint. If the design is found to involve many hazards, a complete redesign may be preferable to patching the original design. [Leveson and Stolzy 1987]

D. ADDING FAILURES TO THE ANALYSIS

In an embedded system control usually cannot be abandoned abruptly, therefore, responses to hardware failures, software faults, human error, and undesired and perhaps unexpected environmental conditions must be built into the system.

'According to Leveson and Stolzy [1987], these responses can take three basic forms:

- 1) a fault tolerant system continues to provide full performance and functional capabilities in the presence of operational faults.

- 2) a fail-soft system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed.
- 3) a fail-safe system attempts to limit the amount of damage caused by failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

These responses are basically in the order of decreasing desirability although the functional and safety requirements of the system are not identical, they are not necessarily of decreasing importance [Leveson and Stolzy 1987].

To prove the safety of a complex system in the presence of faults, it is necessary to show that no single fault can cause a hazardous effect and that hazards resulting from sequences of failures are sufficiently remote. The latter approaches the impossible if an attempt is made to combine all possible failures in all possible sequences and to analyze the output. Instead, procedures start by defining what is hazardous and then working backward to find all combinations of faults that produce the event. [Leveson and Stolzy 1987].

Having reduced the risk of the system to an acceptable level, to continue the analysis it is necessary to consider run-time faults and failures. Designing for fault tolerance and safety requires being able to model failures and faults and to analyze the resulting model. Using definitions from

Kopetz [1982], a failure is defined as an event while a fault is a state. A failure always results in a fault and is called a fault-starting event. The fault remains in the system until a terminating event occurs for the fault. Control failures are the only failures we are concerned with.

According to Leveson and Stolzy [1987], control failures include:

- 1) a required event that does not occur
- 2) an undesired event
- 3) an incorrect sequence of required events
- 4) two incompatible events occurring simultaneously
- 5) timing failures in event sequences

When dealing with analysis of failures in general situations, it is useful to be able to determine the state that a system is in after the failure has occurred. A fault remains in the system until a terminating event for the fault (the faulty condition is no longer true or loses its token). Because of the faulty state or condition, it is possible for further failures to occur that cause further faults.

With failure insertion, we can insert a failure at almost any conceivable place in the model. A realistic failure condition must be considered when determining failures at places. For more information on techniques to ensure

fail-safe or fault-tolerant operation see [Leveson and Stolzy 1987].

E. SUMMARY OF ANALYSIS RESULTS

The results of this analysis shows that the software in this system and in systems in general must monitor the system environment better.

From the analysis and from discussions with the designer of the system, three improvements to the system are apparent.

- 1) Need to verify 4+G level, before continuing to detonation
- 2) Add telemetry data about the state of the device, in its toggle phases. This check could be done every 1,000 feet.
- 3) Add a watchdog monitor to address all critical timing constraints.
- 4) Before stopping the software make sure the software is no longer needed for monitoring.

In order to perform a thorough analysis, the analyst needs to be aware of all types of potential failures, mechanical, computer hardware and software.

E. SUMMARY OF ANALYSIS METHODOLOGY

The situations that were examined in this chapter show that the techniques and concepts presented here and by Leveson

and Stolzy [1987] are appropriate for software safety analysis. Untimed Petri nets are able to provide adequate information to show certain types of analysis and Timed Petri nets could provide additional timing information in time critical systems.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis has proposed a methodology for modeling a real-time system for the purpose of software safety analysis. This thesis shows that Petri net modeling is sufficient for performing this type of analysis. Petri net modeling allows the analyst to focus on the essentials of the system in regard to software safety without worrying about unnecessary details. This modeling method provides a realistic abstraction of actual features of the real-time system.

This methodology is not a substitute for a well-trained analyst. The translation from the design documentation to the Petri net model can be automated in addition to automated tools to assist in the analysis, but an expert analyst will always be needed. If a safety hazard is left out or not considered in the analysis the system is potentially unsafe. Any amount of automation of this process will not point out safety design problems by itself. The technique proposed, given the hazardous conditions, can be used to see if these can occur and if they do, either remove them or limit their effect.

The construction of the model and analysis does not attempt to address the issue of the correctness of the

software or the correctness of the design, but it does allow analysis of the existing design.

This tool and methodology is in its infancy. With increased interest this design/analysis tool can at least begin to approach questions of software safety.

In conjunction with the work performed by Leveson, hopefully this work can provide the springboard to show that the field of software safety is an approachable field. A benefit of this research is it provides an incentive for people to begin using this type of tool in their own applications and to further this area of research.

B. RECOMMENDATIONS

Since this field is at a beginning period, more work is needed to develop automated tools to build the Petri net model and automated tools to perform some of the analysis. An automated tool which can aid the creation of a net and allow queries for unsafe states would provide immeasurable time savings. This might provide an ability to question graphs and perform traces. Without the presence of automated tools this type of analysis would be impractical to perform on even a small system.

One of the concerns is that when this type of work is scaled-up the amount of time or effort will not be

proportional to the time and effort of this size project.

This question cannot yet be answered, but further research is needed to answer that question.

Since the analysis in this thesis did not use Time Petri nets, an application of Time Petri nets would be important to determine the effect of this methodology. The addition of timing constraints will clearly increase the level of the analysis of the system analyzed in this thesis.

This thesis has primarily focused on the creation of the Petri net model in order to evaluate the concepts necessary for the analysis. To provide a final design for the safety arming device, a full-scale analysis would need to be performed.

APPENDIX A SOFTWARE CODE LISTING

```
* CURRENT SAFE SEPARATION SENSOR DESIGN *
;
;
;*****
;
* PORT ASSIGNMENTS *
;*****
;
;
; Port 0 (bus) is the ADC input
; Port 1 is the control port
; bits 0,1 ADC Control
; bit 2 display control * not used *
; bits 3,4,5,6 solenoid control
; bit 7 switch input (ITL)
; Port 2 is the display output * not used *
;
;*****
;
* VARIABLE/REGISTER ASSIGNMENTS *
;*****
;
;
SOLSTAT EQU R0 ; SOLENOID STATUS
REFX EQU R1 ; LOOK-UP TABLE POINTER
VELL EQU R2 ; LOWER VELOCITY BYTE
VELH EQU R3 ; UPPER VELOCITY BYTE
SEPDISL EQU R4 ; LOWER SEP. DISTANCE BYTE
SEPDISH EQU R5 ; UPPER SEP. DISTANCE BYTE
ACLREF EQU R6 ; INITIAL ACCELERATION
FSTREF EQU 60 ; LOWER ADDRESS OF LOOK-UP TABLE
REFMAX EQU 63 ; UPPER ADDRESS OF LOOK-UP TABLE
;
;*****
;
;
; ORG 00H ; GOTO MEMORY LOC 00
;*****
;
; MOV A,#87H ;
; OUTL P1,A ; INITIALIZE P1 *
;*****
;
;*****
;
; MOV R1,#REFMAX ;
; MOV @R1,#020H ;
; DEC R1 ;
; MOV @R1,#040H ; BUILD LOOK-UP TABLE FOR SOLENOID *
; DEC R1 ; TOGGLE DETERMINATION *
; MOV @R1,#080H ;
; DEC R1 ;
; MOV @R1,#0C0H ;*****
;
;
;
```

INIT:	MOV A,#FSTREF	*****
	MOV R1,A	;* LOAD LOW LOOK-UP TABLE ADDRESS *
		;* INTO R1 (REFX) *

		;
		;
	CLR A	*****
	MOV R0,A	;* *
	MOV R2,A	;* *
	MOV R3,A	;* *
	MOV R4,A	;* CLEAR STORAGE REGISTERS *
	MOV R5,A	;* *
	MOV R6,A	;* *

		;
		;

		;* CHECK SWITCH SUBROUTINE *

		;
CKSWCH:	IN A,P1	;* READ PORT 1 DATA INTO ACCUMULATOR
	JB7 FIRED	;* JUMP IF BIT 7 NOT ZERO (SWITCH ON)
	CALL RDACCL	;* CALL ADC READ SUBR

	CPL A	;* UPDATE INITIAL ACCELERATION, TWO'S *
	INC A	;* COMPLEMENT FORMAT *
	MOV ACLREF,A	*****
		;
	JMP CKSWCH	;* CHECK AGAIN FOR SWITCH 'ON'

		;* FIRED SUBROUTINE *

		;
FIRED:	CALL RDACCL	;* CALL ADC READ SUBR
	ADD A,ACLREF	;* ACCOUNT FOR INITIAL ACCELERATION
	CLR C	;* CLEAR CARRY

	ADD A,VELL	;* *
	MOV VELL,A	;* *
	CLR A	;* UPDATE CURRENT VELOCITY *
	ADDC A,VELH	;* *
	MOV VELH,A	;* *
	MOV A,VELL	*****
		;

		;* *
	ADD A,SEPDISL	*****
	MOV SEPDISL,A	;* *

	MOV A,VELH	;*	UPDATE CURRENT SEP. DISTANCE	*
	ADDC A,SEPDISH	;		
	MOV SEPDISH,A	;		
		;	*****	
	JC TOGGLE	;	JUMP IF CARRY ON SEP. DISTANCE	
	CALL KILLTME	;	CALL TIME DELAY SUBR	
		;	*****	
	MOV A,SEPDISH	;		
	ADD A,@REFX	;	COMPARE SEP. DISTANCE W/LOOKUP	
TABLE		;		
	JNC FIRED	;	JUMP IF TOGGLE NOT REQUIRED	
	JMP TOGGLE	;	JUMP TO TOGGLE SUBR	
		;		
		;	*****	
		;	READ ADC SUBROUTINE	*
		;	*****	
		;		
		;		
RDACCL:	ANL P1,#0FEH	;	ENABLE ADC	
	ANL P1,#0FCH	;	LOAD ACCELERATION ONTO BUS	
	INS A,BUS	;	INPUT CURRENT ACCELERATION	
	ORL P1,#3H	;	DISABLE ADC	
	RET	;	RETURN TO CALLING PROGRAM	
		;	*****	
		;	TOGGLE SOLENOID SUBROUTINE	*
		;	*****	
		;		
		;		
TOGGLE:	MOV A,SOLSTAT	;	LOAD SOLENOID STATUS INTO ACC	
	INC SOLSTAT	;	TOGGLE BIT 0 OF SOLENOID STATUS	
	INC REFX	;	INCREMENT LOOK-UP TABLE POINTER	
		;	*****	
	JB0 CCW	;		*
	ORL P1,#0D7H	;	TOGGLE SOLENOID ACCORDING TO BIT 0*	*
	JMP WAIT	;	OF SOLENOID STATUS	*
CCW:	ORL P1,#0AFH	;		*
		;	*****	
		;		
		;	*****	
WAIT:	MOV A,#0C8H	;	LOAD AND START TIMER, WAIT FOR	*
	CALL STWAIT	;	SOLENOID TO TOGGLE	*
		;	*****	
		;	*****	
	MOV A,#087H	;	DISABLE SOLENOID, ADC	*
	OUTL P1,A	;		*
		;	*****	
		;		
	MOV A,REFX	;	*****	

ADD A,#0C1H	; CHECK LOOK-UP TABLE, JUMP IF	*
JNC FIRED	; LAST VALUE HAS NOT BEEN EXCEEDED	*

	;	
	;	

	* WAIT FOR SWITCH OFF SUBROUTINE	*

	;	
	;	

SWOFF: IN A,P1	* WAIT FOR SWITCH TO BE TURNED OFF,	*
JB7 SWOFF	* POSSIBLE RESTART OF PROGRAM	*
JMP INT	*****	
	;	
	;	

	* TIMER CONTROL SUBROUTINE	*

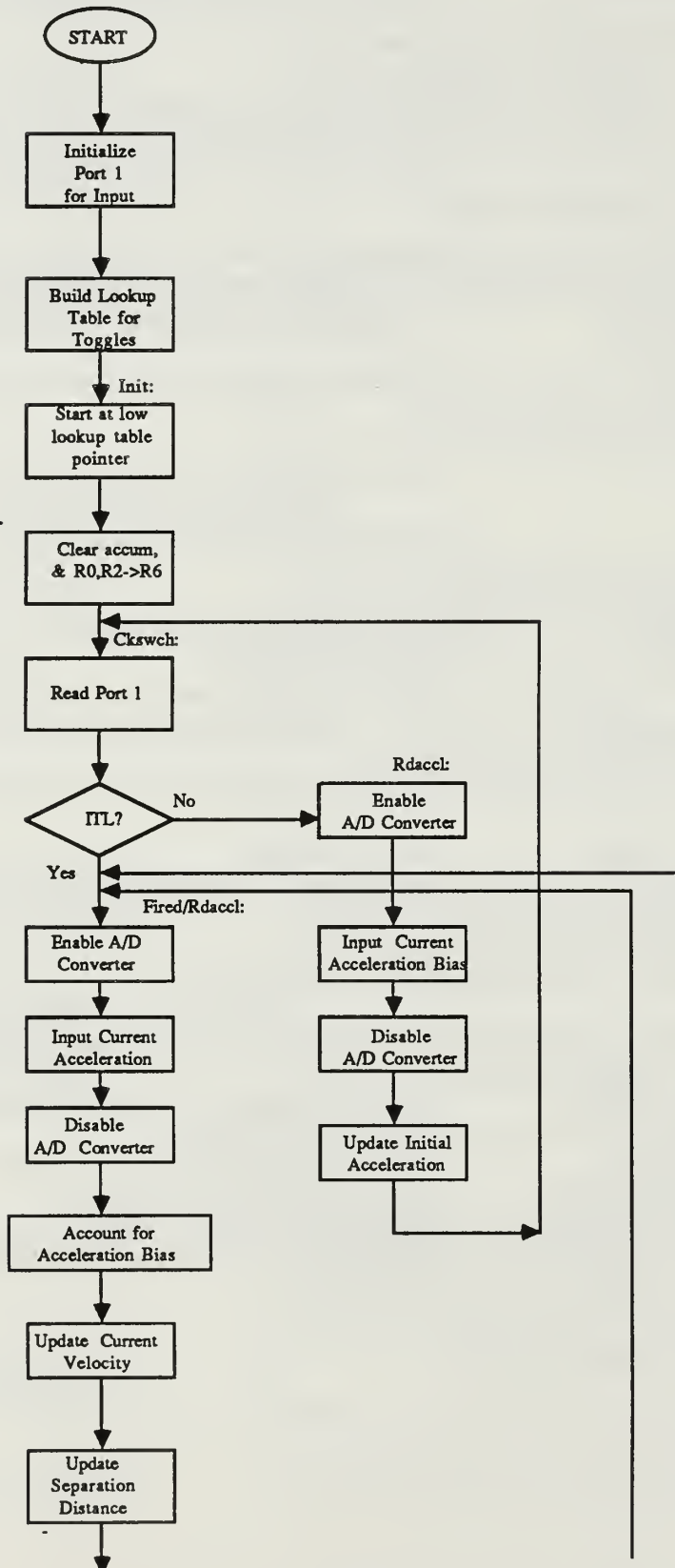
	;	
	;	
STWAIT: MOV T,A	; LOAD STARTING TIME	
STRT T	; START TIMER	

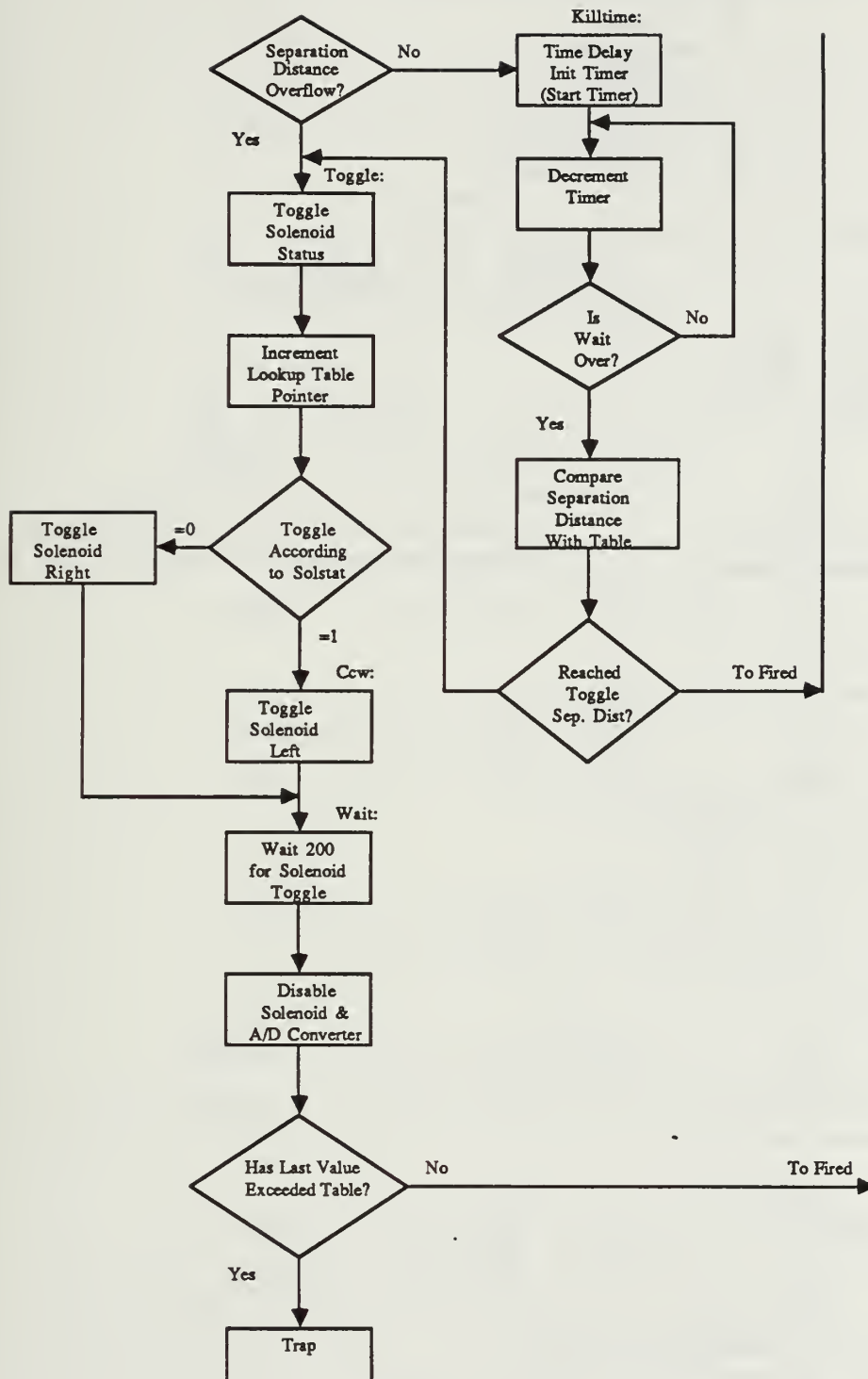
TIMER: JTF WAITDON	* WAIT FOR TIMER TO COMPLETE	*
JMP TIMER	*****	
	;	
	;	
WAITDON: STOP TCNT	; STOP TIMER	
RET	; RETURN TO CALLING PROGRAM	
	;	

	* TIME DELAY SUBROUTINE	*

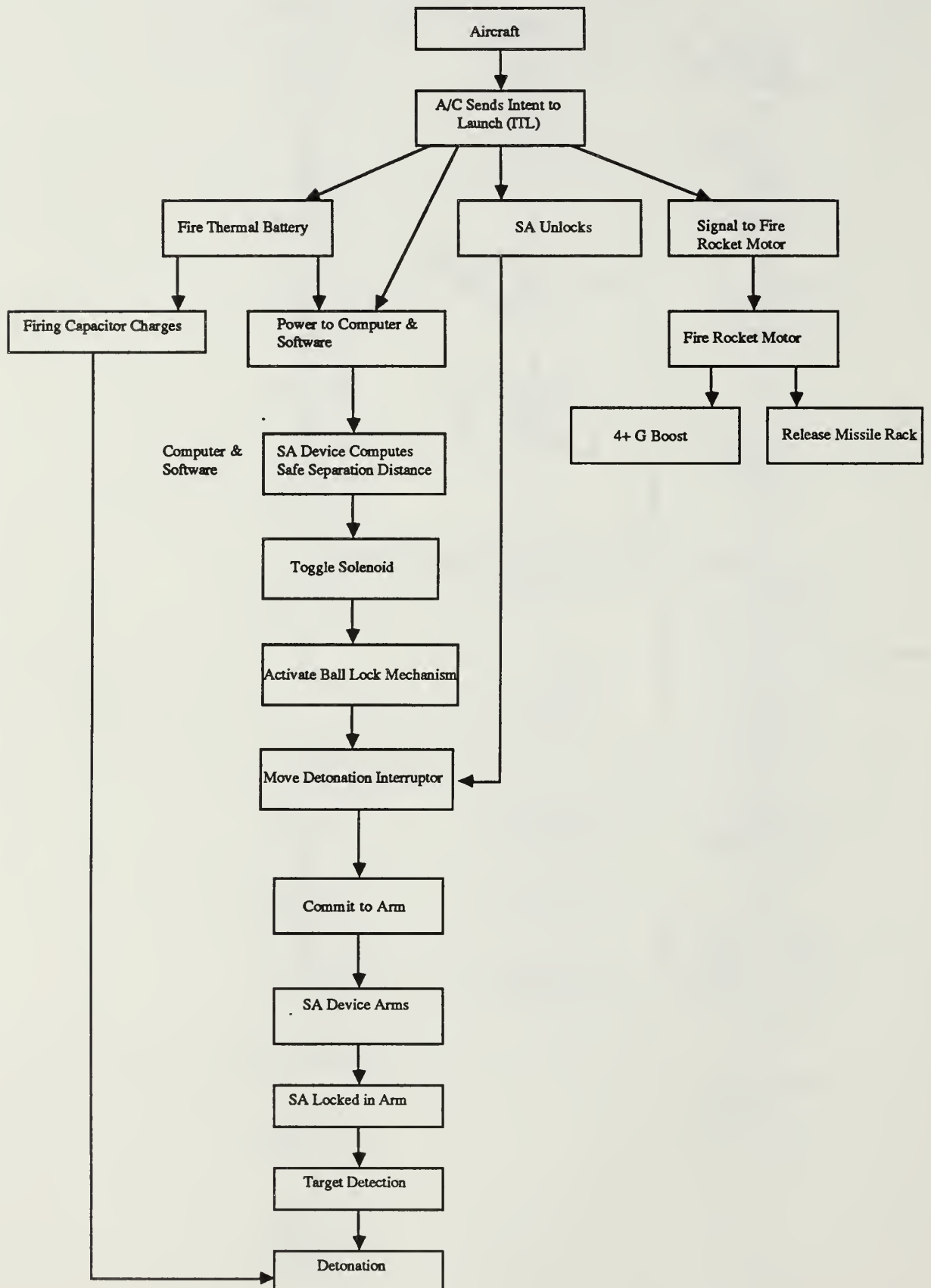
	;	
KILLTME: MOV A,#11H	; LOAD STARTING TIME INTO ACC	
LOOPTME: DEC A	; DECREMENT ACC	
JZ RETRN	; IF WAIT COMPLETE, RETURN	
CALL STWAIT	; CALL TIME DELAY SUBR	
JMP LOOPTME	; RESTART TIMER	
RETRN: RET	; RETURN TO CALLING PROGRAM	
	;	
	;	
END		

APPENDIX B SOFTWARE FLOWCHART

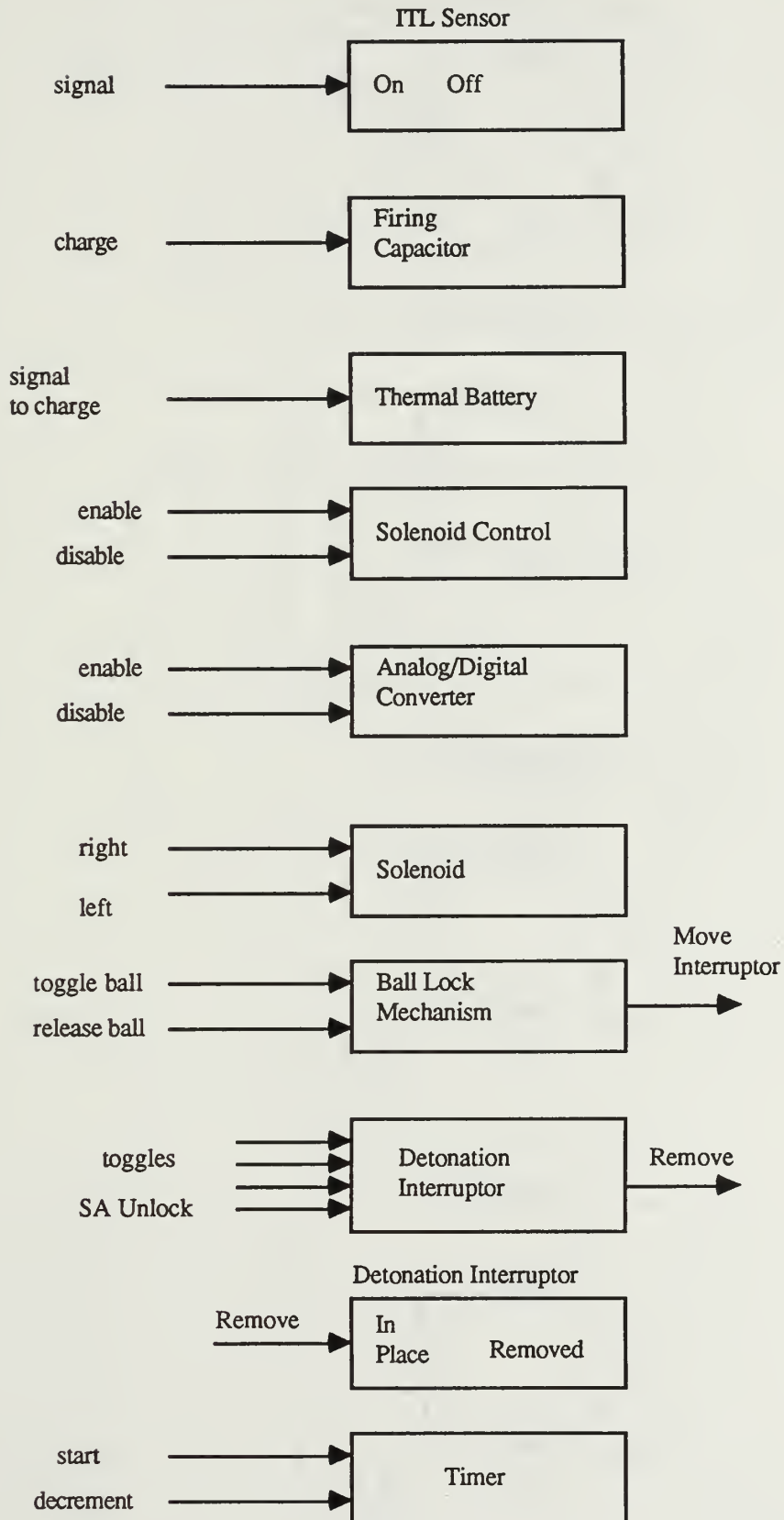




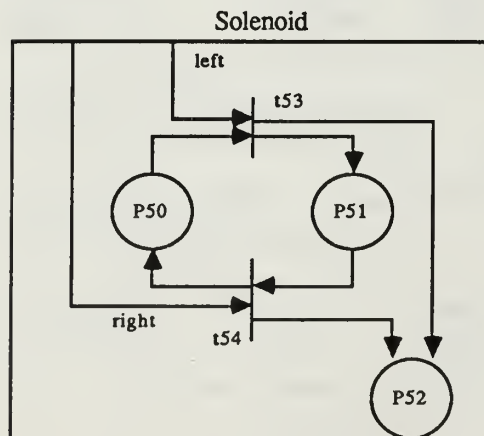
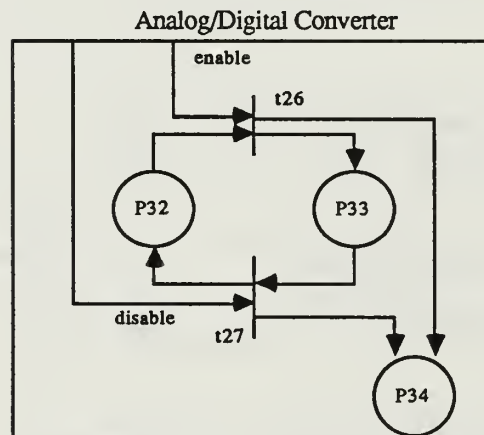
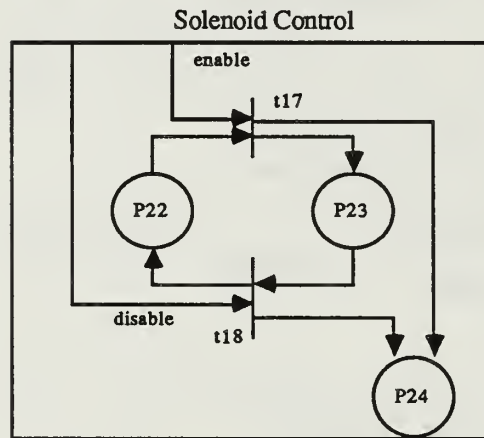
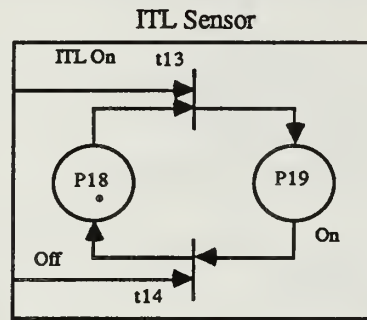
APPENDIX C SYSTEM FLOWCHART



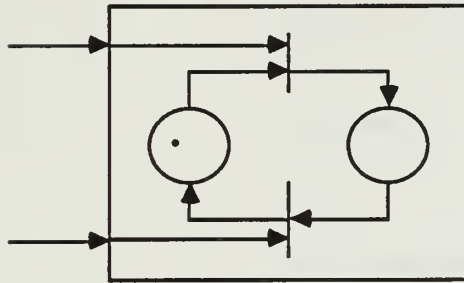
APPENDIX D. SYSTEM ELEMENTS



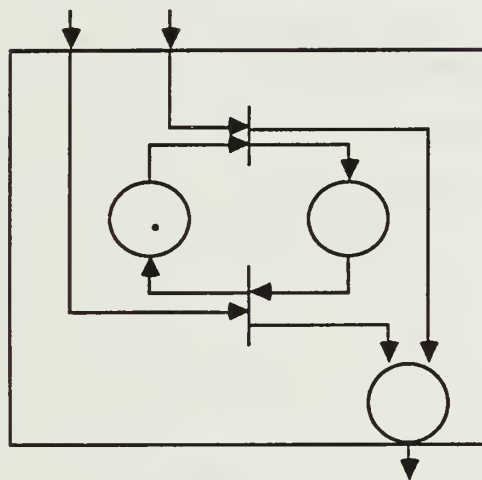
APPENDIX E PETRI NET OF SYSTEM ELEMENTS



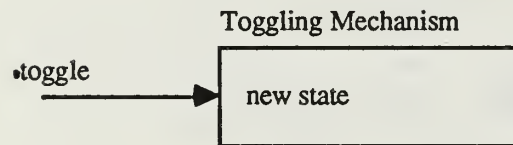
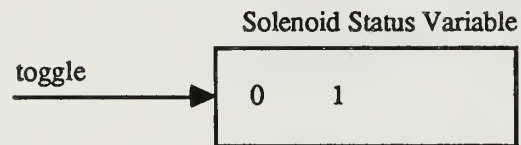
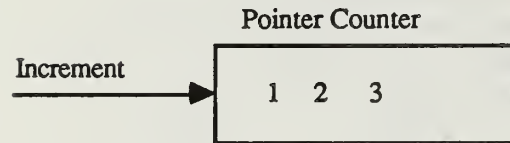
Two-State Element Without Response



Two-State Element With Response



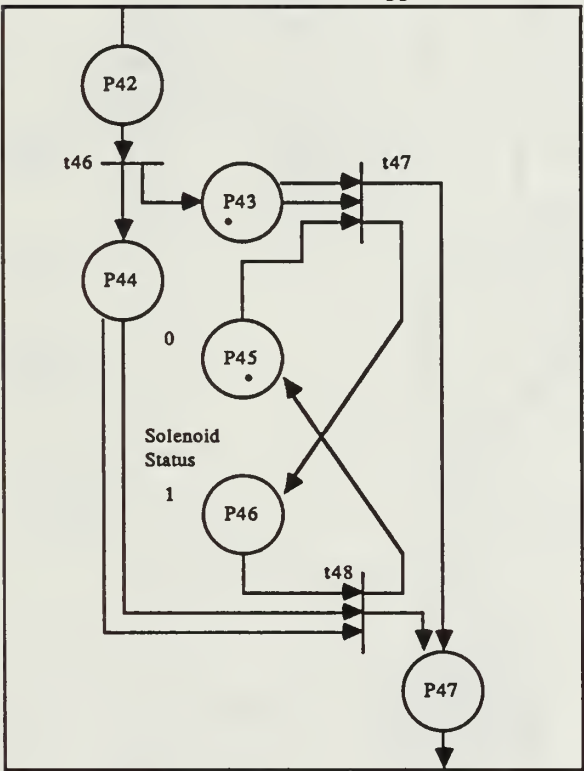
APPENDIX F SOFTWARE ELEMENTS



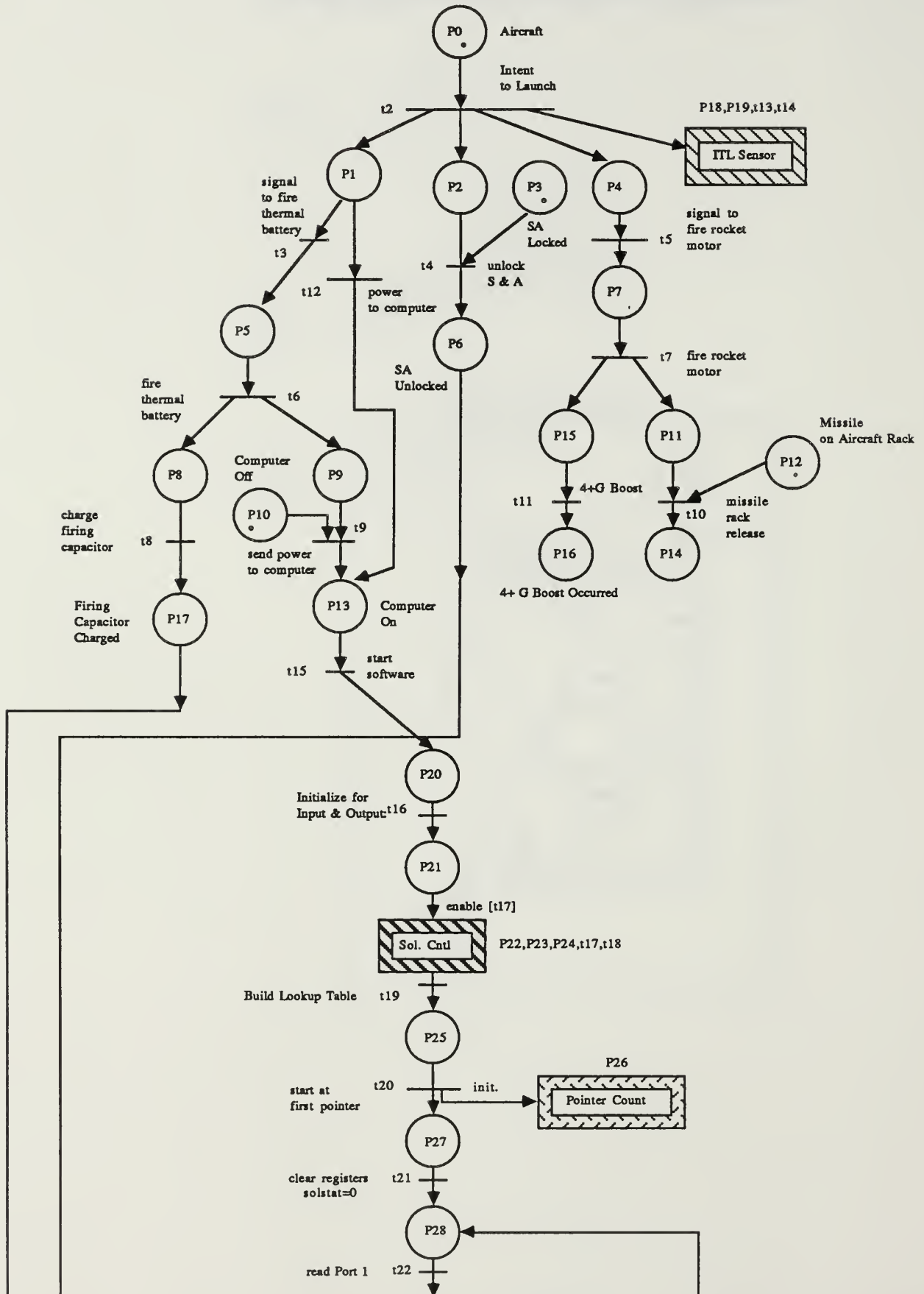
Pointer Counter

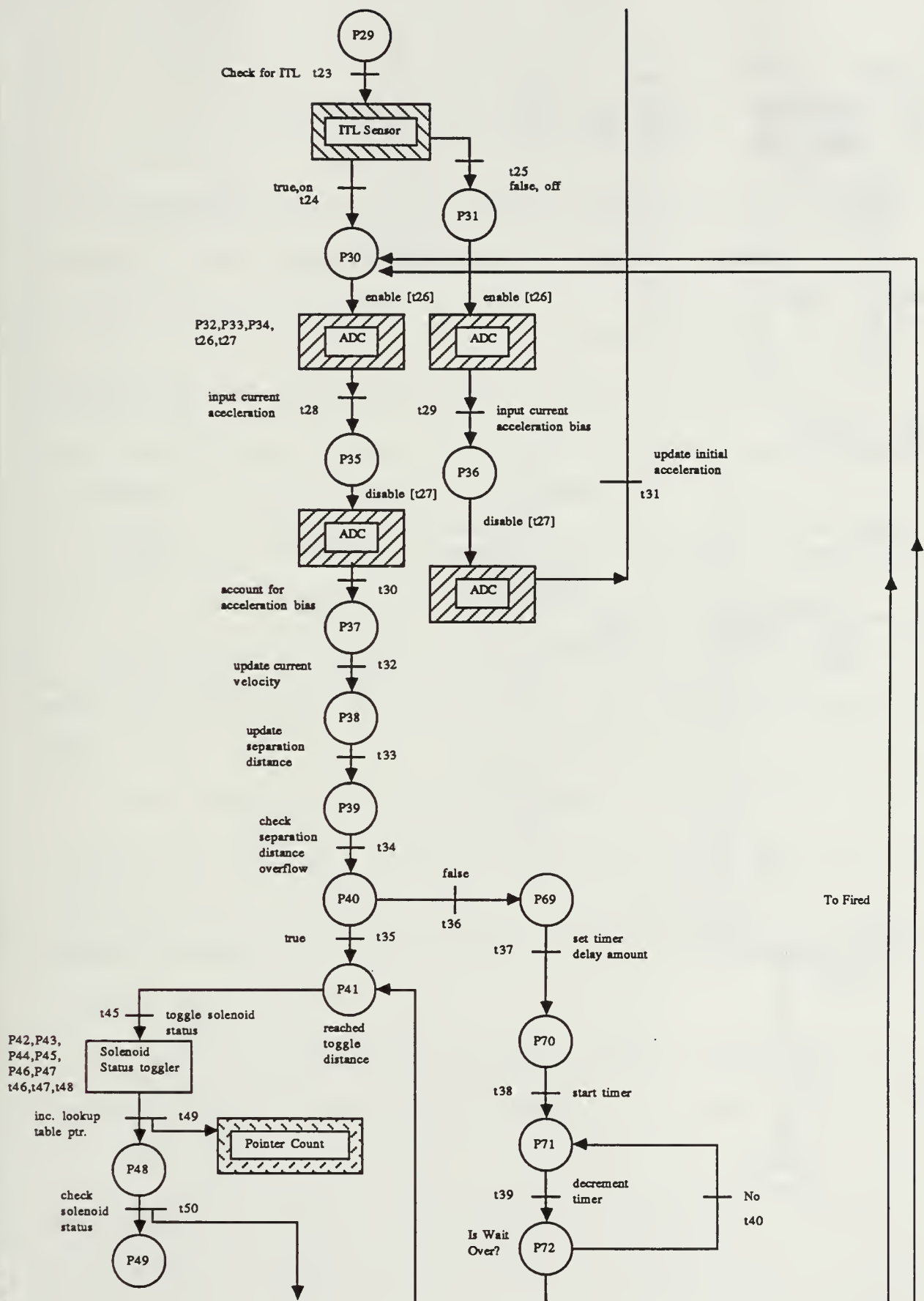


Solenoid Status Toggler



APPENDIX H PETRI NET MODEL





APPENDIX I REACHABILITY THEORY

The state of a Petri net is defined by its markings. The change in state caused by firing a transition is defined by the next-state function δ .

definition: The next state function $\delta: N^n \times T \rightarrow N^n$ for a Petri net $\Phi = (P, T, I, O, \mu_0)$ with marking μ and transition $t_j \in T$ is defined if and only if t_j is enabled.

If $\delta(\mu, t_j)$ is defined, then $\delta(\mu, t_j) = \mu'$ where $\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$ for all $p_i \in P$.

Given an initial state, the reachability set for the Petri net is the set of states that results from executing the Petri net.

Given a transition sequence and μ_0 (initial mark), we can easily derive the marking sequence for the execution of the Petri net. In a marking μ , a set of transitions will be enabled and may fire. The result of firing a transition in a marking μ is a new marking μ' .

We say μ' is immediately reachable from μ ; that is we can immediately get to state μ' from state μ .

definition: a marking μ' is immediately reachable from μ if there exists a transition $t_j \in T$ such that $\delta(\mu, t_j) = \mu'$.

The reachability relationship is the reflexive transitive closure of the immediately reachable relationship.

Thus if μ' is immediately reachable from μ , and μ'' is immediately reachable from μ' , then μ' is reachable from μ .

definition: The reachability set $R(\Phi, \mu)$ for a Petri net $\Phi = (P, T, I, O, \mu_0)$ with marking m is the smallest of markings defined by:

- 1) $\mu \in R(\Phi, \mu)$
- 2) If $\mu' \in R(\Phi, \mu)$ and $\mu' = \delta(\mu', t_j)$, for some $t_j \in T$, then $\mu'' \in R(\Phi, \mu)$.

APPENDIX J CRITICAL STATE DETERMINATION ALGORITHM

The following is the critical state determination algorithm as proposed by [Leveson and Stolzy 1987]:

```
Put initial set of high-risk conditions into S = states_to_process
while S is not empty
do
  let c be one of S;
  if c is a subset of the initial state then
    high-risk state reachable and need to redesign
  else
    do { work backwards to critical states }
      next_back_states =  $\emptyset$ 
      for each transition  $t \in T$  {determine which transitions are
        enabled}
        do
          let  $R = O(t) \cap c$ ;
          if  $R \neq \emptyset$  then {t is enabled, generate the corresponding
            next backward states}
            Next_back_states = Next_back_states  $\cup \delta^{-1}(R \cup (O(t) - R) \cup (c - R), t)$ ;
        od
      for each next_back_state b
        do
          forward_states =  $\emptyset$ 
          for each transition  $t \in T$  {determine which transitions
            are enabled}
            do
```

```

    let  $R = I(t) \cap b$ ;

    if  $R \neq \emptyset$  then {t is enabled, generate the
corresponding forward states }

        forward_states = forward_states  $\cup \delta (R \cup (I(t) - R)$ 
 $\cup (b - R), t)$ ;

    od

Other_states = Forward_states - [Forward_states  $\cap$ 
(S  $\cup$  Next_back_states)]

case b

    b  $\in$  states_considered: exit;

    b is illegal according to system invariants: exit;

    b is high risk and there exists  $f \in$  Other_states such
        that f is low-risk {therefore b is potentially
critical}: add b to set of critical states;

    else {b is low-risk but not critical-necessary to go
backwards again}

        add b to S;

    esac

od

move c from S to states_considered;

augment design by eliminating bad transition paths from
critical states;

od

od

```

LIST OF REFERENCES

- Anderson, T. and Lee, P.A., Fault Tolerance: Principles and Practice, Prentice-Hall, Englewood Cliffs, N.J., 1981.
- Avizienis, A., The N-version approach to fault tolerant software. IEEE Transactions on Software Engineering, SE-11,12 Dec. 1985, p. 1491-1501.
- Berthomieu, B. and Menasche, M., An enumerative approach for analyzing time Petri nets, Proceedings of 1983 IFIP Congress, Paris, Sept. 1983.
- Boebert, W. E., Formal verification of embedded software. ACM Software Engineering Notes 56,3, July, 1980, p. 41-42.
- Dean, E.S. Software system safety. Proceedings of the 5th International System Safety Conference (Denver, Col.), vol. 1, part 1, System Safety Society, Newport Beach, Calif., p. III-A-1 to III-A-8.
- Dijkstra, E.W., Cooperating Sequential Processes in Programming Languages, Genuys, F, London Academic Press, 1965.
- Dennis, J.B. and Van Horn E.C., Programming Semantics for Multiprogrammed Computations, Communications of the ACM, vol. 9, p 143-155, Mar. 1966.
- Ericson, C. A. Software and system safety. Proceedings of the 5th International System safety Conference (Denver, Col.), vol. 1, part 1, System Safety Society, Newport Beach, Calif., p. III-B-1 to III-B-1, 1981.
- Frola, F. R. and Miller, C.O., System Safety in Aircraft Management, Logistics Management Institute, Washington, D.C., Jan. 1984.
- Gloe, G., Inspection of process computers for nuclear power plants. Proceedings of IFAC SAFECOMP '79. Pergamon, Elmsford, N.Y. p. 213-218. 1979.
- Griggs, J.G. A method of software safety analysis. Proceedings of the Safety Conference (Denver, Col.), vol 1, part 1. System Safety Society, Newport Beach, Calif. p. III-D-1 to III-D-18., 1981.
- Intel Corp., MCS-48 Microcomputer User's Manual, 1978.

- Jahanian, F. and Mok, A. K., Safety analysis of timing properties in real-time systems. IEEE Transactions on Software Engineering, SE-12, 9 Sept. 1986, p. 890-904.
- Johnson, W. G., The management oversight and risk tree. MORT, U.S. Atomic Energy Commission, SAN 821-2, UC-41, 1973.
- Kletz, T., Human problems with computer control. Hazard Prevention, Mar./Apr. 1983, p. 24-26.
- Konakovsky, R. Safety evaluation of computer hardware and software. Proceedings of Compsac '78, IEEE, New York, p. 559-564, 1978.
- Kopetz, H. The failure fault model, Proceedings FTCS-12, Santa Monica, Calif., June 1982, p.14-17.
- Lauber, R., Strategies for the design and validation of safety-related computer-controlled systems. Real-time Data Handling and Process Control, G. Meyer, Ed. North-Holland Publishing., Amsterdam, p. 305-310, 1980.
- Leveson, N. G., and Stolzy, J. L. , Safety Analysis of Ada Programs Using Fault Trees, IEEE Transactions on Reliability, Vol. R-32, No. 5, Dec. 1983.
- Leveson, N. G., Software Safety: Why, What, and How, Computing Surveys, Vol. 18, No.2, June 1986.
- Leveson, N. G., and Stolzy, J. L., Safety Analysis Using Petri Nets, IEEE Transactions on Software Engineering, Vol SE-13, No. 3, Mar. 1987.
- Malasky, S. W., System Safety Technology and Application, Garland STPM Press, New York, 1982.
- McIntee, J. W., Fault Tree Technique as Applied to Software (Soft Tree), BMO/AWS, Norton Air Force Base, Calif., 1983.
- McVay, J., Point paper on conducting a design, development, and safety review of a guided missile safety-arming device utilizing a noninterrupted explosive train., NWC TM (draft), NWC, China Lake, Calif., 1987.
- Merlin, P.M. and Farber D.J., Recoverability of communication protocols-Implications of a theoretical study, IEEE Transactions on Communications, Vol. COM-24, p. 1036-1043, Sept. 1976.
- MIL-STD-1316C, Safety Criteria for Fuze Design, Dept. of Defense, Wash., D.C., 3 Jan. 1984.

- MIL-STD-1574A (USAF) System Safety Program for Space and Missile Systems, , Dept of Air Force, Government Printing Office, Wash. D.C. Aug. 15 1979.
- MIL-STD-882B Notice 1,, System Safety Program Requirements, U.S. Department of Defense, U.S. Government Printing Office, Wash. D.C., July 1 1987.
- MIL-STD-SNS (Navy), Software nuclear safety (draft), Available from Naval Weapons Evaluation Facility, Kirtland AirForce Base, N.M., 1986.
- Neumann, P. G., Letter from the Editor., ACM Software Engineering Notes 6, 2, 1981.
- Perrow, C., Normal Accidents: Living with High Risk Technologies, Basic Books New York. 1984.
- Petersen, D., Techniques of Safety Management, McGraw-Hill, New York, 1971.
- Peterson, J.L., Petri Net Theory and the Modeling of Systems, Englewood Cliffs, N.J., Prentice-Hall, 1981.
- Petri, C., Kommunikation mit Automaten, Ph. D. Dissertation, Univeristy of Bonn, Bonn, West Germany, 1962.
- Roland, H. E. and Moriarity, B., System Safety Engineering and Management, Wiley, N.Y., 1983.
- Software Safety Handbook (Draft). H.Q. AFISC/SESD, Norton Air Force Base, Calif. 92409.
- Vesely, W.E., Goldberg, F. F., Roberts, N. H., Haasl, D. F., Fault Tree Handbook, U. S. Nuclear Regulatroy Commission, Report NURTEG-0492, Jan. 1981.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3. Commander (Code 34) Naval Air Test Center Patuxent River, Maryland 20670	2
4. Commander (Code 31C) Naval Weapons Center China Lake, California 93555	2
5. Commander (Code 3353) Naval Weapons Center China Lake, California 93555	5
6. Daniel Davis (Code 52DV) Naval Postgraduate School Monterey, California 93943-5002	3
7. Duston L. Hayward c/o John A. Hayward 537 West 217 Street New York, New York 10034	3

Thesis
H4075
c.1

Hayward

A practical application
of Petri nets in the
software safety analysis
of a real-time military
system.

Thesis
H4075
c.1

Hayward

A practical application
of Petri nets in the
software safety analysis
of a real-time military
system.

thesH4075

A practical application of Petri nets in



3 2768 000 76891 5

DUDLEY KNOX LIBRARY